Homomorphic Encryption

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

FHE:
2. Background

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

Polynomials

$$Enc_k(A \circ B) = Enc_k(A) \circ Enc_k(B)$$

With lattices, we use polynomials to represent our multi-dimensional spaces. In general, a polynomial can be represented by a number of coefficients (a_n ... a_0) and polynomial powers:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + ...a_1 x + a_0 \tag{2.33}$$

Within a lattice, we can have a point at (9, 5, 16), and then represent it with a quadratic equation of:

$$16x^2 + 5x + 9 \tag{2.34}$$

$$f = (16x^2 + 5x + 9)(2x^2 + x + 7) = 32x^4 + 26x^3 + 135x^2 + 44x + 63 \tag{2.35}$$

$$\begin{aligned} A \times B &= (10x^2 + 6x + 2) \times (12x^2 + 3x + 2) \\ &= 120x^4 + 30x^3 + 20x^2 + 72x^3 + 180x^2 + 12x + 24x^2 + 6x + 4 \\ &= 120x^4 + 102x^3 + 62x^2 + 18x + 4 \end{aligned}$$

And now, we can apply a $\pmod{13}$ operation to each of the coefficients:

$$A \times B = 120x^4 + 102x^3 + 62x^2 + 18x + 4 = 3x^4 + 11x^3 + 10x^2 + 5x + 4 \pmod{13}$$

Inv Mod p

2.3.2 Inverse polynomial mod p

With lattice methods, we use the shortest vector problem in a lattice, which has an underpinning difficulty of factorizing polynomials. The lattice vector points are represented as polynomial values. For this, we create a modulo polynomial and then generate its inverse. In this case, we will create a polynomial (f) and then find its modulo inverse (f_p), and which will result in:

$$f \cdot f_p = 1 \pmod{p} \tag{2.45}$$

Thus, if we then take a message (m) and multiply it by f and then by f_p, we should be able to recover the message. Let's take an example with N=11 (the highest polynomial factor), p=31 and where Bob picks polynomial factors (f) of:

$$f = [-1, 1, 1, 0, -1, 0, 1, 0, 0, 1, -1] \tag{2.46}$$

$$f(x) = -1x^{10} + 1x^9 + 1x^6 - 1x^4 + 1x^2 + 1x - 1 \pmod{31} \tag{2.47}$$

We then determine the inverse of this with:

$$f_p : [9, 5, 16, 3, 15, 15, 22, 19, 18, 29, 5] \tag{2.48}$$

$$fp(x) = 5x^{10}+29x^9+18x^8+19x^7+22x^6+15x^5+15x^4+3x^3+16x^2+5x+9 \pmod{31} \tag{2.49}$$

RLWE

Learning with errors is a method defined by Oded Regev in 2005 [43] and is known as LWE (Learning With Errors). It involves the difficulty of finding the values which solve:

$$\mathbf{B} = \mathbf{A} \times \mathbf{s} + \mathbf{e} \tag{2.52}$$

where you know $\mathbf{A}$ and $\mathbf{B}$. The value of $\mathbf{s}$ becomes the secret values (or the secret key), and $\mathbf{A}$ and $\mathbf{B}$ can become the public key A video is here [44].

$$b_A = \begin{bmatrix} 4 & 1 & 11 & 10 \\ 5 & 5 & 9 & 5 \\ 3 & 9 & 0 & 10 \\ 1 & 3 & 3 & 2 \\ 12 & 7 & 3 & 4 \\ 6 & 5 & 11 & 4 \\ 3 & 3 & 5 & 0 \end{bmatrix} \times \begin{bmatrix} 6 \\ 9 \\ 11 \\ 11 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 1 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} \pmod{13} = \begin{bmatrix} 4 \\ 7 \\ 2 \\ 11 \\ 5 \\ 12 \\ 8 \end{bmatrix}$$

$$\mathbf{A} = a_{n-1}x^{n-1} + \ldots + a_1 x + a_1 x^2 + a_0 \tag{2.53}$$

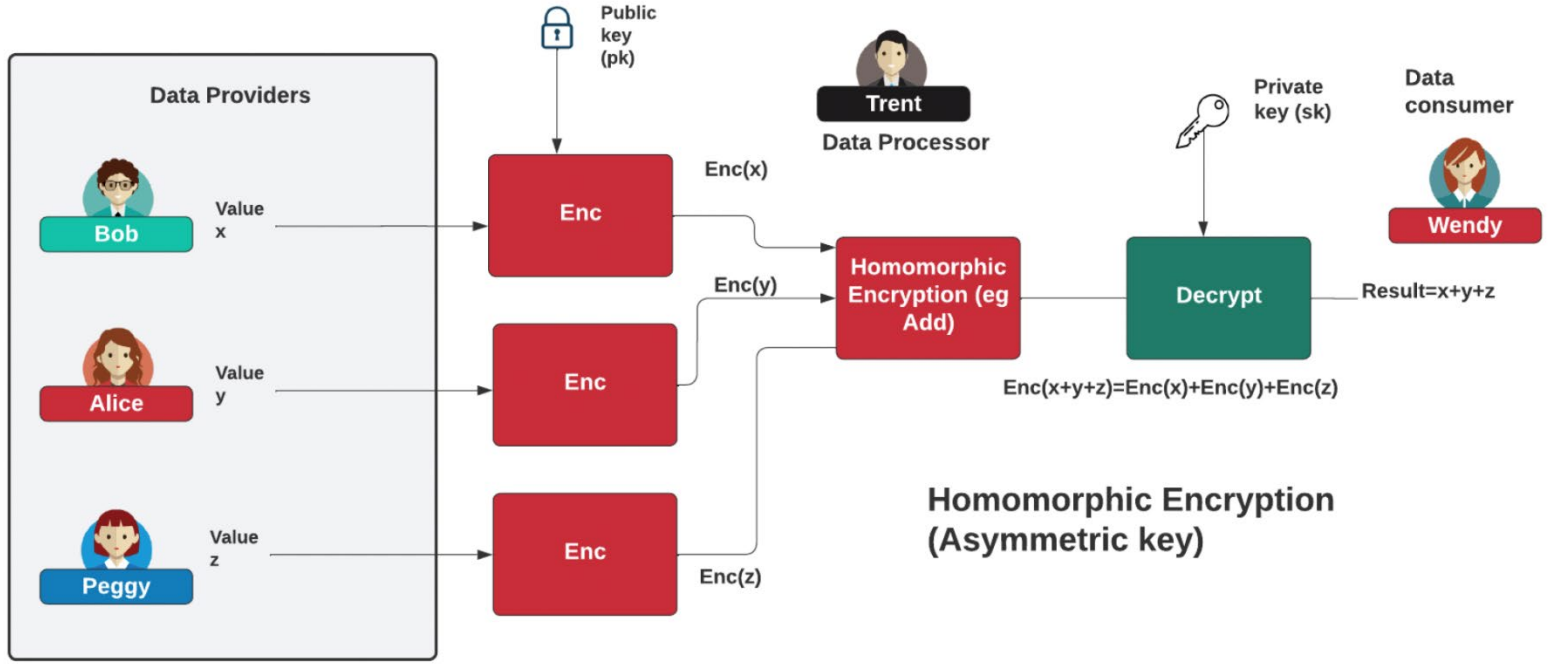
Next Alice will divide by $\Phi(x)$, which is $x^n + 1$:

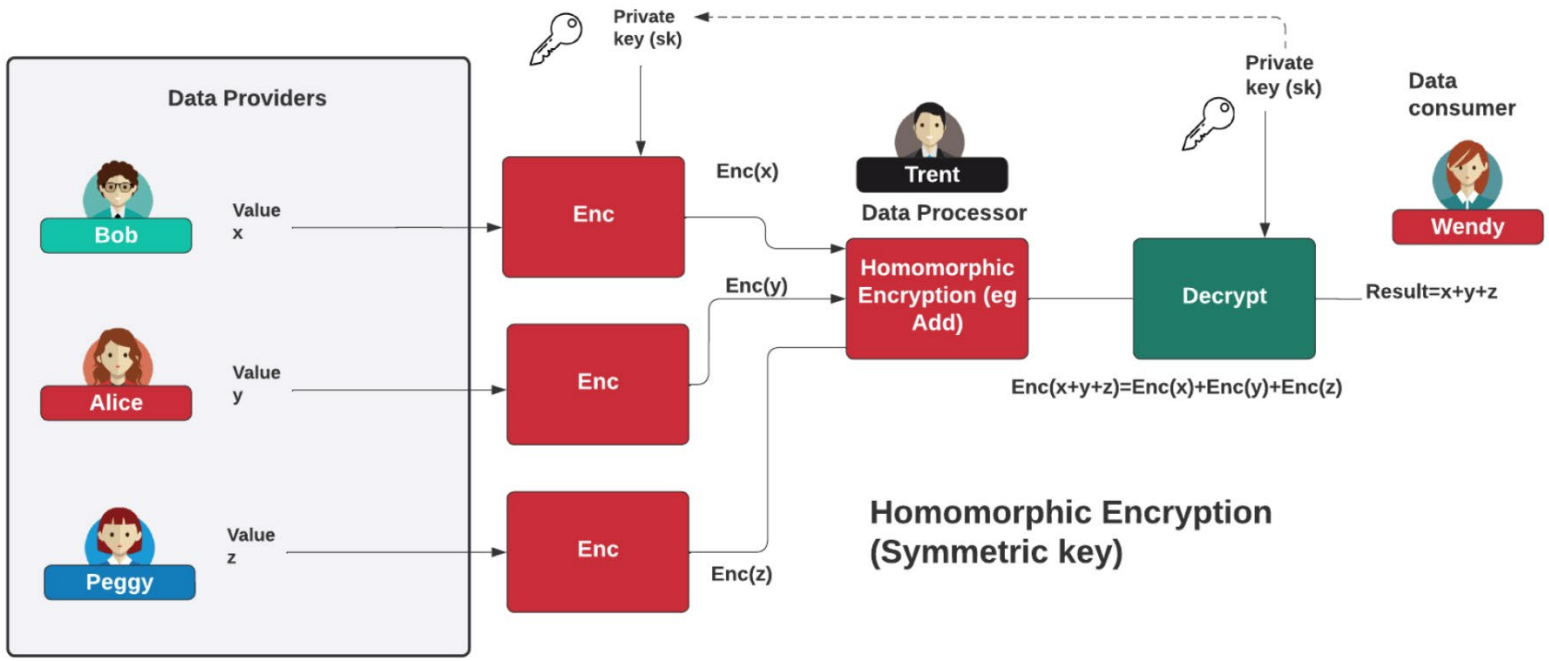
$$\mathbf{A} = (a_{n-1}x^{n-1} + \ldots + a_1 x + a_1 x^2 + a_0) \div (x^n + 1) \tag{2.54}$$

FHE

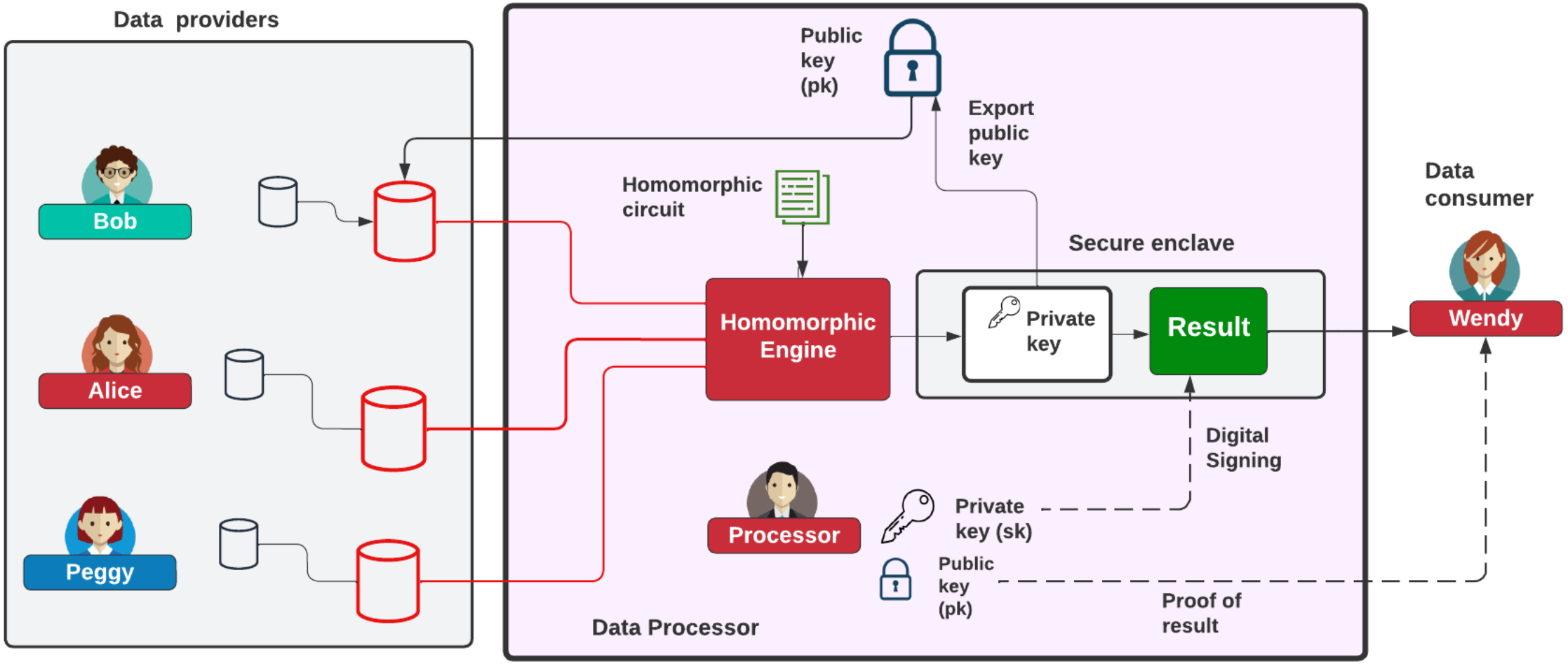
- 1st generation: Gentry's method uses integers and lattices [50] including the DGHV method.
- 2nd generation. Brakerski, Gentry and Vaikuntanathan's (BGV) and Brakerski/Fan-Vercauteren (BFV) use a Ring Learning With Errors approach [51]. The methods are similar to each other, and with only small difference between them.
- 3rd generation: These include DM (also known as FHEW) and CGGI (also known as TFHE) and support the integration of Boolean circuits for small integers.
- 4th generation: CKKS (Cheon, Kim, Kim, Song) and which uses floating-point numbers [52].

FHE

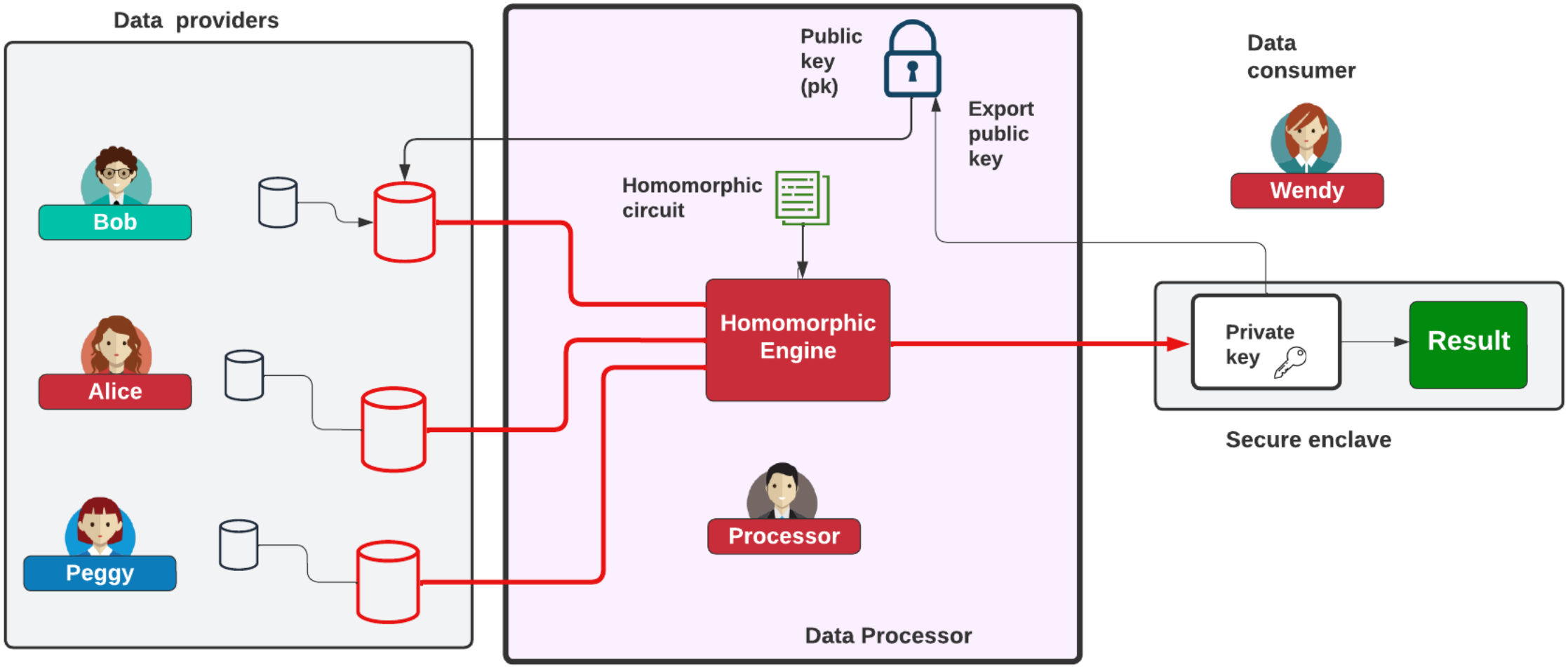

Public key (pk)
Data Providers
Bob
Value x
Alice
Value y
Peggy
Value z
Enc
Enc
Enc
Enc(x)
Enc(y)
Enc(z)
Trent
Data Processor
Homomorphic Encryption (eg Add)
Private key (sk)
Decrypt
Result=x+y+z
Enc(x+y+z)=Enc(x)+Enc(y)+Enc(z)
Data consumer
Wendy
Homomorphic Encryption (Asymmetric key)


Private key (sk)
Private key (sk)
Data Providers
Bob
Value x
Alice
Value y
Peggy
Value z
Enc
Enc
Enc
Enc(x)
Enc(y)
Enc(z)
Trent
Data Processor
Homomorphic Encryption (eg Add)
Decrypt
Result=x+y+z
Enc(x+y+z)=Enc(x)+Enc(y)+Enc(z)
Data consumer
Wendy
Homomorphic Encryption (Symmetric key)

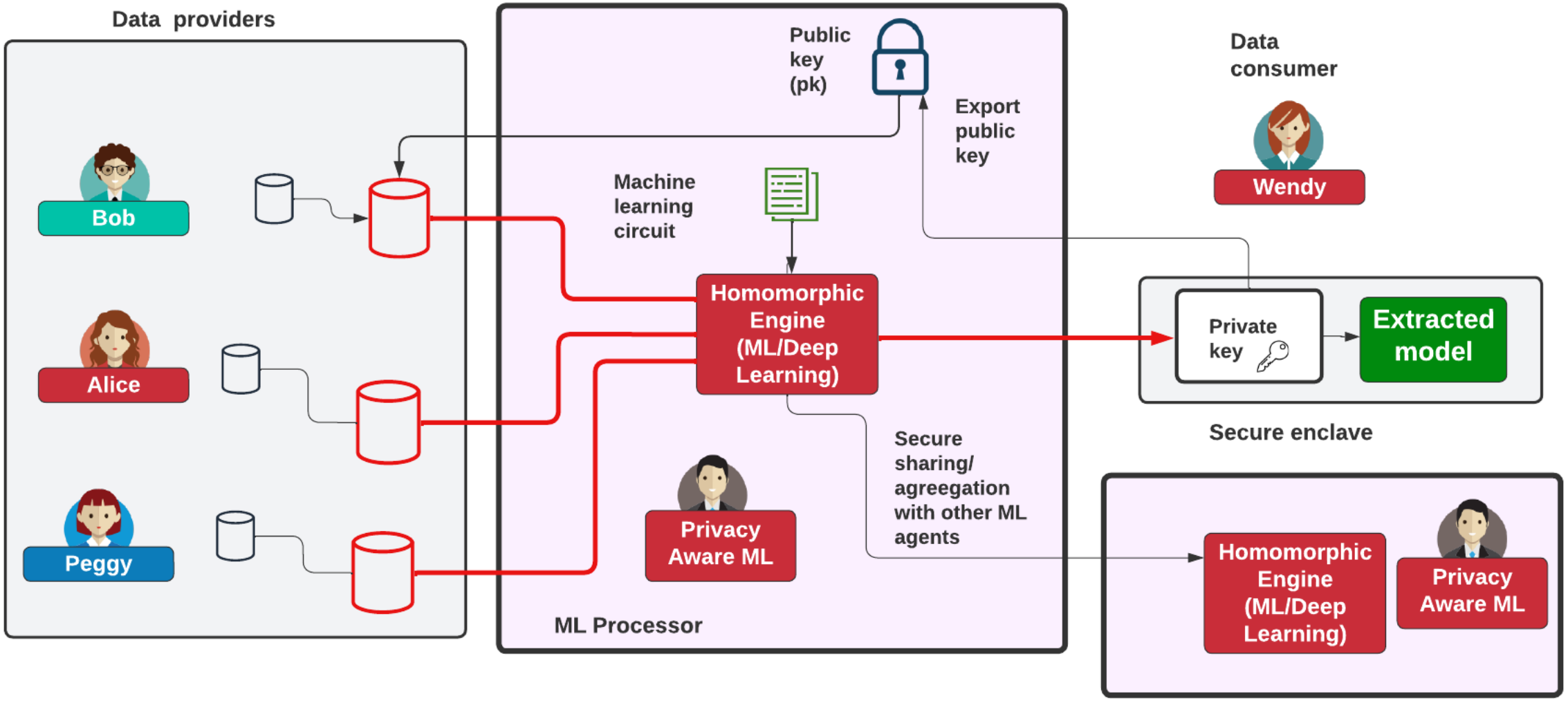
Homomorphic Processing



Homomorphic Processing



Homomorphic ML



Cryptography: 3. State-of-the-art

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

Libraries

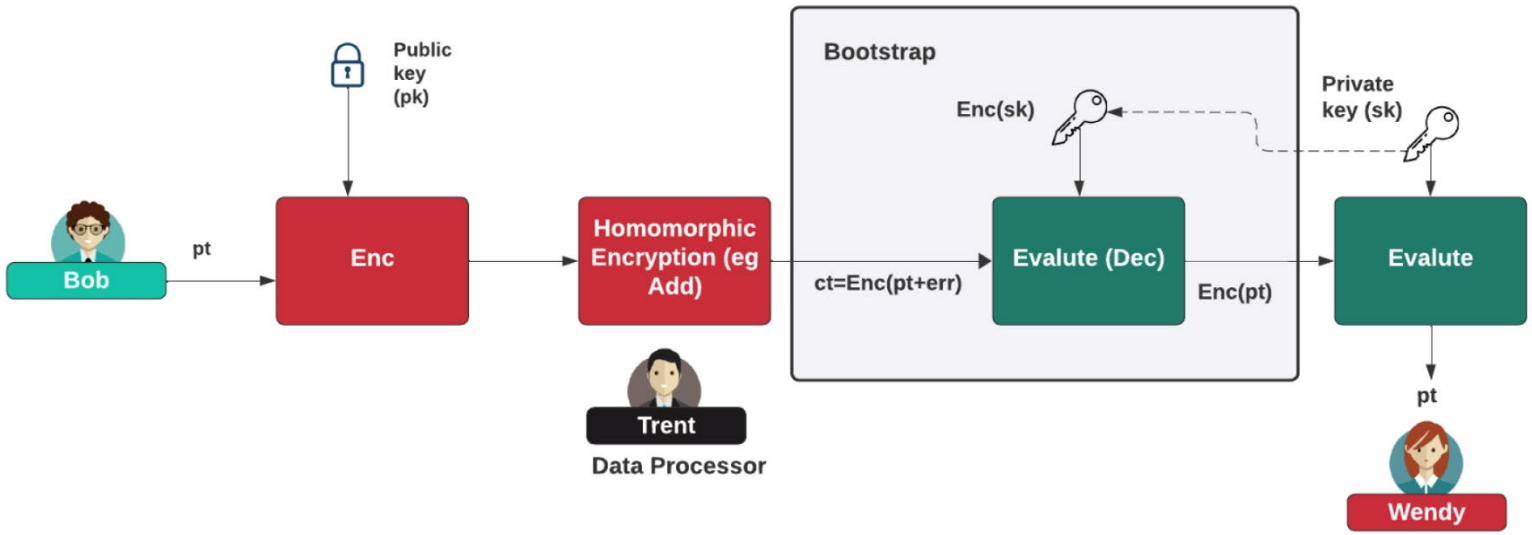
Product	Creator	Language	License	Summary
SEAL [177]	Microsoft	C++	MIT	Widely-used FHE library that implements BFV for modular arithmetic and CKKS for approximate arithmetic.
HElib [116]	IBM	C++	Apache-2.0	Widely-used FHE library that implements BGV for modular arithmetic and CKKS for approximate arithmetic.
TFHE [59]	Gama et al.	C++	Apache-2.0	Implements an optimized ring variant of the GSW scheme.
HEAAN [115]	CryptoLab, Inc.	C++	CC-BY-NC-3.0	Implements the CKKS approximate number arithmetic scheme.
PALISADE [165]	New Jersey Institute of Technology	C++	BSD-2-Clause	Lattice cryptography library that supports multiple protocols for FHE, including BGV, BFV, and StSt.
$\Lambda \circ \lambda$ [69]	E. Crockett and C. Peikert	Haskell	GPL-3.0-only	Pronounced "LOL." Implements a BGV-type FHE scheme.
Cingulata [45]	CEA LIST	C++	CECILL-1.0	Compiler and RTE for C++ FHE programs. Implements BFV and supports TFHE.
FV-NFLlib [89]	CryptoExperts	C++	GPL-3.0-only	Implements FV scheme. Built on the NFLLib lattice cryptography library. Last updated 2016.
Lattigo [138]	Laboratory for Data Security	Go	Apache 2.0	Implements BFV and HEAAN in Go.

OpenFHE

OpenFHE

- Brakerski/Fan-Vercauteren (**BFV**) scheme for integer arithmetic
- Brakerski-Gentry-Vaikuntanathan (**BGV**) scheme for integer arithmetic
- Cheon-Kim-Kim-Song (**CKKS**) scheme for real-number arithmetic (includes approximate bootstrapping)
- Ducas-Micciancio (**DM**) and Chillotti-Gama-Georgieva-Izabachene (**CGGI**) schemes for Boolean circuit evaluation.

Bootstrapping



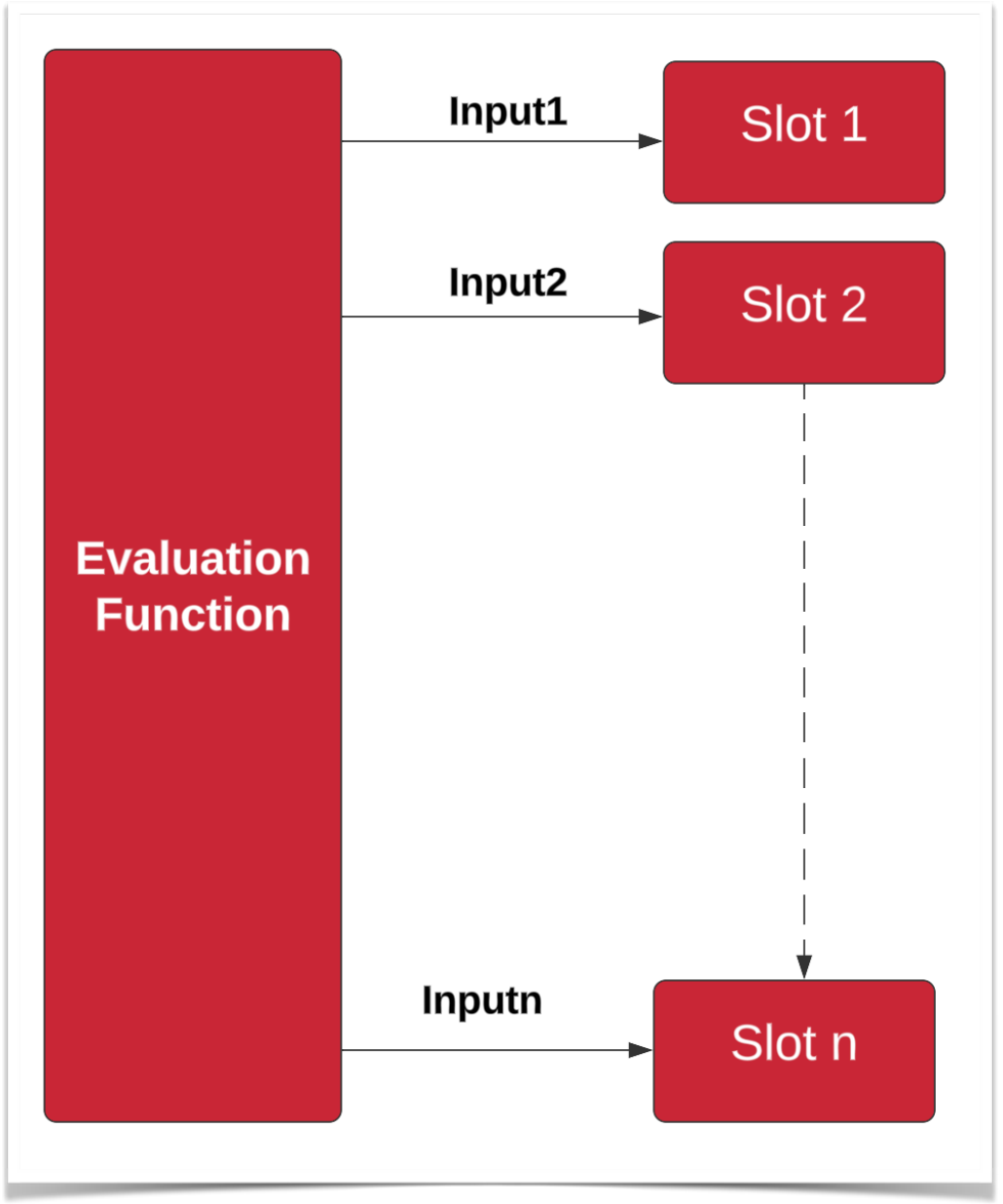
The main bootstrapping methods are CKKS [52], DM [57]/CGGI, and BGV/BFV. Overall, CKKS is generally the fastest bootstrapping method, while DM/CGGI is efficient with the evaluation of arbitrary functions. These functions approximate maths functions as polynomials (such as with Chebyshev approximation). BGV/BFV provides reasonable performance and is generally faster than DM/CGGI but slower than CKKS.

Bootstrapping and Slots

Each ciphertext can have an associated "level" and a value of "noise".

We have various levels. One multiplication consumes a level, and adds noise.

When we reach zero or the noise level is too high, we need to bootstrap - raises the level and reduce noise.



BGV/BFV

With BGV and BFV, we use a Ring Learning With Errors (LWE) method [51]. With BGV, we define a moduli (q), which constrains the range of the polynomial coefficients. Overall, the methods use a moduli, which can be defined within different levels. We then initially define a finite group of $\mathbb{Z}_q$, and then make this a ring by dividing our operations with $(x^n + 1)$ and where $n-1$ is the largest power of the coefficients. The message can then be represented in binary as:

$$m = a_{n-1}a_{n-2}...a_0 \tag{3.1}$$

This can be converted into a polynomial with:

$$\mathbf{m} = a_{n-1}x^n a_{n-1} + a_{n-2}x^{n-2} + ... + a_1 x + a_0 \pmod{q} \tag{3.2}$$

The coefficients of this polynomial will then be a vector. Note that for efficiency, we can also encode the message with ternary (such as with -1, 0 and 1). We then define the plaintext modulus with:

$$t = p^r \tag{3.3}$$

and where p is a prime number and r is a positive number. We can then define a ciphertext modulus of q, and which should be much larger than t. To encrypt with the private key of $\mathbf{s}$, we implement:

$$(c_0, c_1) = \left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s} + e, -\mathbf{a}\right) \mod q \tag{3.4}$$

BGV/BFV

To decrypt:

$$m = \lfloor \frac{t}{q}(c_0 + c_1).\mathbf{s} \rceil \tag{3.5}$$

This works because:

$$\begin{aligned} m_{recover} &= \lfloor \frac{t}{q}\left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s} + e - \mathbf{a}.\mathbf{s}\right) \rceil && (3.6)\\ &= \lfloor \left(\mathbf{m} + \frac{t}{q}.e\right) \rceil && (3.7)\\ &\approx m && (3.8)\\ & && (3.9) \end{aligned}$$

Parameters and Public Key

3.8.2 Parameters

We thus have a parameter of the ciphertext modulus (q) and the plaintext modulus (t). Both of these are typically to the power of 2. An example of q is 2^{240} and for t is 65,537. As the value of 2^q is likely to be a large number, we typically define it as a log_q value. Thus a ciphertext modulus of 2^{240} will be 240 as defined as a log_q value.

$$\mathbf{pk_1} = (\mathbf{r}.\mathbf{sk} + e) \pmod{q} \tag{3.12}$$

$$\mathbf{pk_2} = \mathbf{r} \tag{3.13}$$

and where r is a random polynomial value. To encrypt with the public key (pk):

$$(c_0, c_1) = \left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s}.\mathbf{r} + e, -\mathbf{a}.\mathbf{r}\right) \mod q \tag{3.14}$$

We then decrypt with the private key (s);

$$m = \lfloor \frac{t}{q}(c_0 + c_1).\mathbf{s} \rceil \tag{3.15}$$

This works because:

$$m_{recovered} = \lfloor \frac{t}{q}\left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s}.\mathbf{r} + e - \mathbf{a}.\mathbf{r}.\mathbf{s}.\right) \rceil \tag{3.16}$$

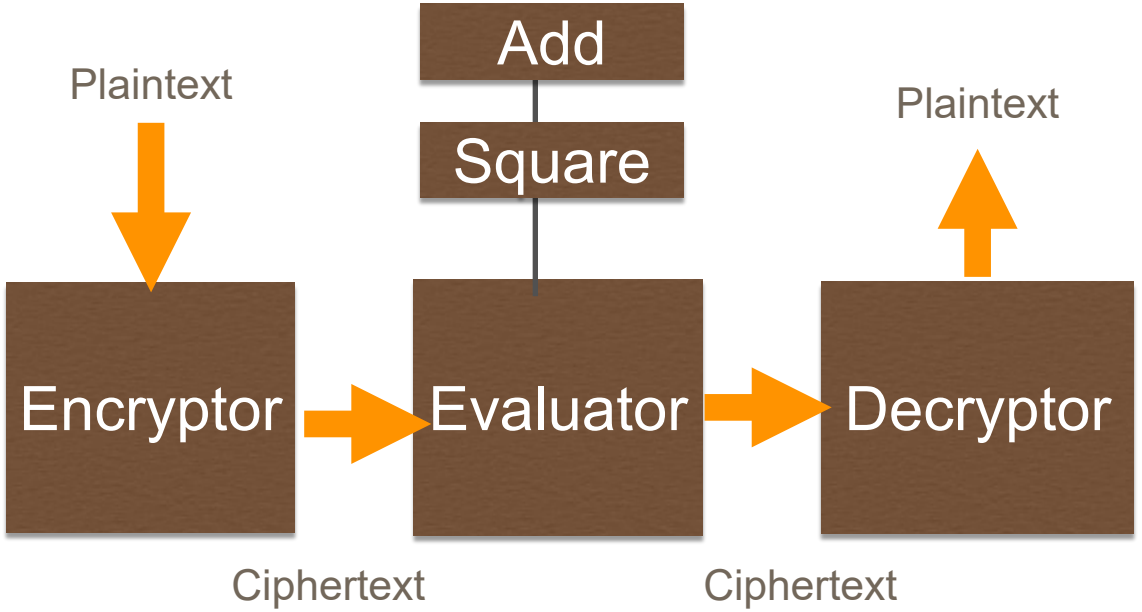
$$= \lfloor \left(\mathbf{m} + \frac{t}{q}.e\right) \rceil \tag{3.17}$$

$$\approx m \tag{3.18}$$

SEAL BFV/BGV

PolyModulusDegree: Defines number of coefficients in plaintext polynomials and the size of ciphertext elements. Must be a power of 2 (such as 1024, 2048, 4096, 8192, 16384, or 32768). Affects performance)

PlainModulus: Largest coefficient that plaintext polynomials can represent. Affects performance.

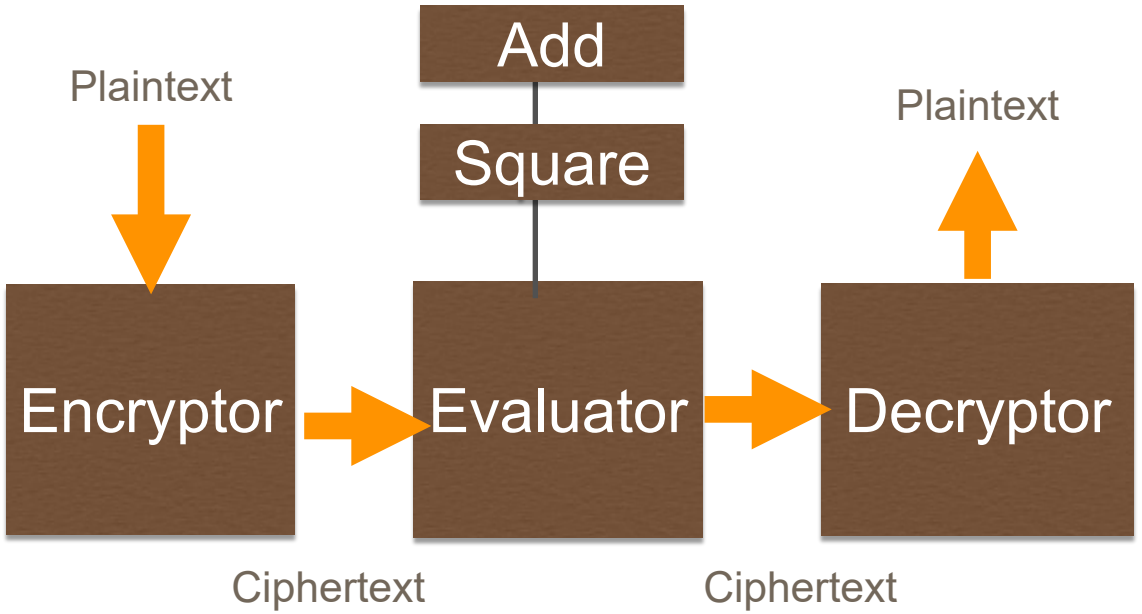



```
EncryptionParameters parms = new EncryptionParameters(SchemeType.BFV);

if (type=="BFV")  parms = new EncryptionParameters(SchemeType.BFV);
else if  (type=="BGV")  parms = new EncryptionParameters(SchemeType.BGV);


Console.WriteLine("Example: {0}\n",type);

ulong polyModulusDegree = mod;
parms.PolyModulusDegree = polyModulusDegree;


parms.CoeffModulus = CoeffModulus.BFVDefault(polyModulusDegree);


parms.PlainModulus = new Modulus(2024);


using SEALContext context = new SEALContext(parms);

using KeyGenerator keygen = new KeyGenerator(context);
using SecretKey secretKey = keygen.SecretKey;
keygen.CreatePublicKey(out PublicKey publicKey);

using Encryptor encryptor = new Encryptor(context, publicKey);
using Evaluator evaluator = new Evaluator(context);
using Decryptor decryptor = new Decryptor(context, secretKey);


using Ciphertext xEncrypted = new Ciphertext();


using Plaintext xPlain = new Plaintext(ULongToString(x));
```

SEAL CKKS

PolyModulusDegree: Defines number of coefficients in plaintext polynomials and the size of ciphertext elements. Must be a power of 2 (such as 1024, 2048, 4096, 8192, 16384, or 32768). Affects performance)

```
using EncryptionParameters parms = new EncryptionParameters(SchemeType.CKKS);


ulong polyModulusDegree = 8192;
parms.PolyModulusDegree = polyModulusDegree;
parms.CoeffModulus = CoeffModulus.Create(
    polyModulusDegree, new int[]{ 60, 40, 40, 60 });


double scale = Math.Pow(2.0, 40);

using SEALContext context = new SEALContext(parms);

Console.WriteLine();

using KeyGenerator keygen = new KeyGenerator(context);
using SecretKey secretKey = keygen.SecretKey;
keygen.CreatePublicKey(out PublicKey publicKey);
keygen.CreateRelinKeys(out RelinKeys relinKeys);
using Encryptor encryptor = new Encryptor(context, publicKey);
using Evaluator evaluator = new Evaluator(context);
using Decryptor decryptor = new Decryptor(context, secretKey);



using CKKSEncoder encoder = new CKKSEncoder(context);
ulong slotCount = encoder.SlotCount;
```



CryptoContext and Parameters (BFV)

Plaintext Modulus

Multiplicative Depth

PKE Scheme Features

Generate key pair (sk, pk)

```
CCParams<CryptoContextBFVRNS> parameters;
parameters.SetPlaintextModulus(mod);
parameters.SetMultiplicativeDepth(2);

CryptoContext<DCRTPoly> cryptoContext = GenCryptoContext(parameters);

cryptoContext->Enable(PKE);
cryptoContext->Enable(KEYSWITCH);
cryptoContext->Enable(LEVELEDSHE);


KeyPair<DCRTPoly> keyPair;

// Generate a public/private key pair
keyPair = cryptoContext->KeyGen();

std::cout << "The key pair has been generated." << std::endl;

auto str = Serial::SerializeToString(  cryptoContext);

cout << "Crypto Context (First 2,000 characters):\n" << str.substr(0,2000) << endl;
```

https://asecuritysite.com/openfhe/openfhe_00cpp

CryptoContext and Parameters (CKKS)

Multiplication depth

Scale Mod Size

```
uint32_t multDepth = 1;
uint32_t scaleModSize = 50;

if (argc>1) {
    std::istringstream iss(argv[1]);
    iss >>scaleModSize;
}

CCParams<CryptoContextCKKSRNS> parameters;
parameters.SetMultiplicativeDepth(multDepth);
parameters.SetScalingModSize(scaleModSize);

CryptoContext<DCRTPoly> cryptoContext = GenCryptoContext(parameters);

cryptoContext->Enable(PKE);
cryptoContext->Enable(KEYSWITCH);
cryptoContext->Enable(LEVELEDSHE);

KeyPair<DCRTPoly> keyPair;

// Generate a public/private key pair
keyPair = cryptoContext->KeyGen();

std::cout << "The key pair has been generated." << std::endl;

auto str = Serial::SerializeToString(  keyPair.publicKey);
```

https://asecuritysite.com/openfhe/openfhe_00cpp_ckks

BFV - Adding/Multiplying Two Numbers

```
// Multiply ciphertext
auto ciphertextMult     = cryptoContext->(ciphertext1, ciphertext2);

// Decrypt result
Plaintext plaintextMultRes;
cryptoContext->Decrypt(keyPair.secretKey,

std::cout << "Method: : " << type << std:
std::cout << "Modulus: : " << mod<< std::

std::cout << "\nx: " << xplaintext << std
```

Encrypt
EvalAdd
EvalAddInPlace
EvalAddMany
EvalAddManyInPlace
EvalAddMutable
EvalAddMutableInPlace
EvalAtIndex
EvalAtIndexKeyGen
EvalAutomorphism
EvalAutomorphismKeyGen
EvalBootstrap

S 2 OUTPUT TERMINAL DEBUG CONSOLE

2024, 12:18:22] Unable to resolve configurat
ng" instead.

```
keyPair = cryptoContext->KeyGen();


std::vector<int64_t>xval = {1};
xval[0]=x;
Plaintext xplaintext                = cryptoContext->MakePackedPlaintext(xval);

std::vector<int64_t> yval = {1};
yval[0]=y;
Plaintext yplaintext                = cryptoContext->MakePackedPlaintext(yval);

// Encrypt values
auto ciphertext1 = cryptoContext->Encrypt(keyPair.publicKey, xplaintext);
auto ciphertext2 = cryptoContext->Encrypt(keyPair.publicKey, yplaintext);

// Add ciphertext
auto ciphertextMult     = cryptoContext->EvalAdd(ciphertext1, ciphertext2);

// Decrypt result
Plaintext plaintextAddRes;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextMult, &plaintextAddRes);
```

https://asecuritysite.com/openfhe/openfhe_02cpp

CCKS - Adding/Multiplying Two Numbers

```
std::vector<double> x1 = {x};
std::vector<double> y1 = {y};

// Encoding as plain
Plaintext ptxt1 = cc
Plaintext ptxt2 = cc

std::cout << "Input
std::cout << "Input

// Encrypt the encod
auto c1 = cc->Encryp
auto c2 = cc->Encryp

//  Addition
auto cAdd = cc->Eval(c1, c2);
//  Subtraction
auto cSub = cc->EvalSub(c1, c2);
// Multiplication
```

```
EvalAdd          inline lbcrypto::Ciphertext<lbcrypto::DCRTPo…
EvalAddInPlace
EvalAddMany
EvalAddManyInPlace
EvalAddMutable
EvalAddMutableInPlace
EvalAtIndex
EvalAtIndexKeyGen
EvalAutomorphism
EvalAutomorphismKeyGen
EvalBootstrap
EvalBootstrapKeyGen
```

```
auto keys = cc->KeyGen();

cc->EvalMultKeyGen(keys.secretKey);

std::vector<double> x1 = {x};
std::vector<double> y1 = {y};

// Encoding as plaintexts
Plaintext ptxt1 = cc->MakeCKKSPackedPlaintext(x1);
Plaintext ptxt2 = cc->MakeCKKSPackedPlaintext(y1);

std::cout << "Input x1: " << ptxt1 << std::endl;
std::cout << "Input y1: " << ptxt2 << std::endl;

// Encrypt the encoded vectors
auto c1 = cc->Encrypt(keys.publicKey, ptxt1);
auto c2 = cc->Encrypt(keys.publicKey, ptxt2);

//  Addition
auto cAdd = cc->EvalAdd(c1, c2);
//  Subtraction
auto cSub = cc->EvalSub(c1, c2);
// Multiplication
auto cMul = cc->EvalMult(c1, c2);

Plaintext result;
std::cout.precision(8);
std::cout << std::endl << "Results: " << std::endl;
cc->Decrypt(keys.secretKey, cAdd, &result);
result->SetLength(batchSize);
std::cout << "x+y=" << result << std::endl;


cc->Decrypt(keys.secretKey, cSub, &result);
result->SetLength(batchSize);
std::cout << "x-y=" <<  result << std::endl;
```

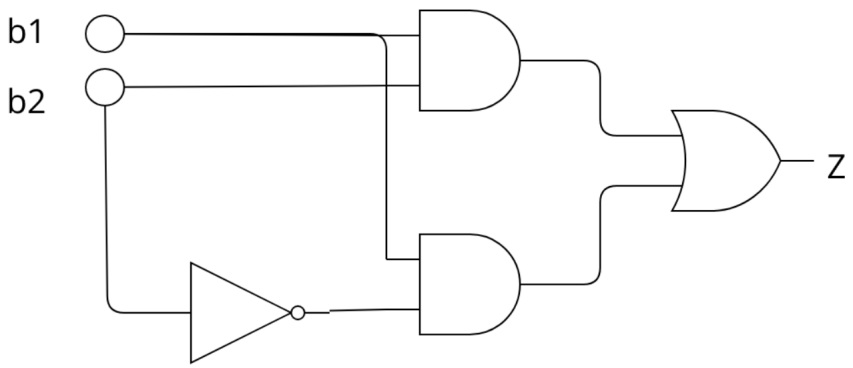
https://asecuritysite.com/openfhe/openfhe_05cpp

BFV - Batch Processing

```
// Generate the relinearization key
cryptoContext->EvalMultKeyGen(keyPair.secretKey);

std::vector<int64_t> xval(count, 0ULL);

for (int i=0;i<count;i++) xval[i]=i;

Plaintext xplaintext   = cryptoContext->MakePackedPlaintext(xval);

// Encrypt values
auto ciphertext1 = cryptoContext->Encrypt(keyPair.publicKey, xplaintext);

// Square
auto ciphertextMult     = cryptoContext->EvalSquare(ciphertext1);

// Decrypt result
Plaintext plaintextAddRes;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextMult, &plaintextAddRes);

std::cout << "Method: : " << type << std::endl;
std::cout << "Parameters " << parameters << std::endl << std::endl;
std::cout << "Ring dimension: " << cryptoContext->GetRingDimension() << "\n";
std::cout << "Modulus: : " << mod<< std::endl;

std::cout << "\nx: " << xplaintext << std::endl;
```

https://asecuritysite.com/openfhe/openfhe_08cpp

MD/FHEW



$$(b_1 . b_2) + (b_1 . \bar{b_2})$$

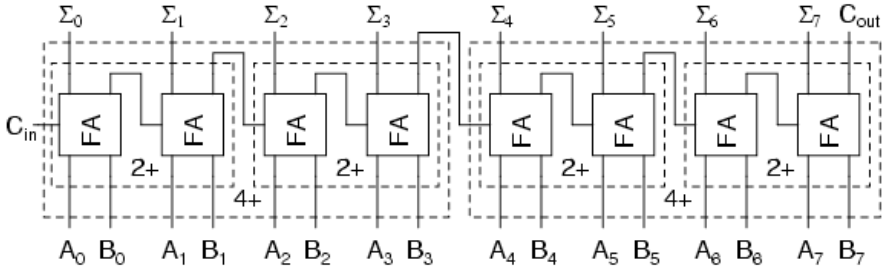
b1	b2	(b1.b2)	(b1.NOT(b2)	Z
0	0	0	0	0
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

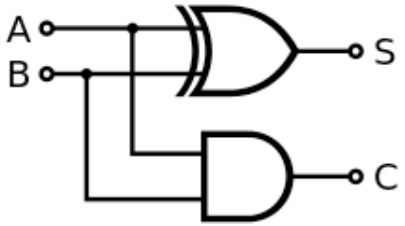
```
auto sk = cc.KeyGen();

std::cout << "Generating the bootstrapping keys... public keys" << std::endl;

// Generate the bootstrapping keys (refresh, switching and public keys)
cc.BTKeyGen(sk, PUB_ENCRYPT);

auto bit1 = cc.Encrypt(cc.GetPublicKey(), b1);
auto bit2 = cc.Encrypt(cc.GetPublicKey(), b2);

cout << bit1 << endl;

auto ctAND1 = cc.EvalBinGate(AND, bit1, bit2);
auto bit2Not = cc.EvalNOT(bit2);
auto ctAND2 = cc.EvalBinGate(AND, bit2Not, bit1);
auto ctResult = cc.EvalBinGate(OR, ctAND1, ctAND2);

LWEPlaintext result;

cc.Decrypt(sk, ctResult, &result);

printf("b1=%d\n",b1);
printf("b2=%d\n",b2);
printf("(b1 AND b2) OR ( b1 AND NOT(b2))\n");
printf("(%d AND %d) OR ( %d AND NOT(%d))=%d\n",b1,b2,b1,b2,result);
```

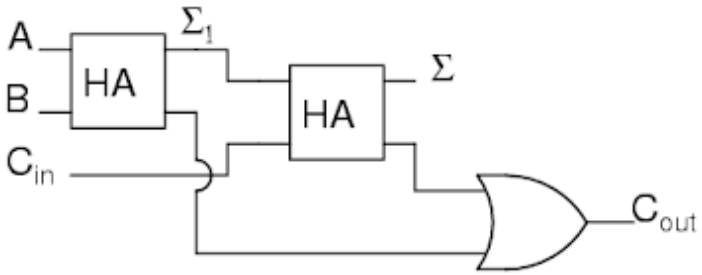
https://asecuritysite.com/openfhe/openfhe_09cpp

https://asecuritysite.com/openfhe/openfhe_09cpp_pke

MD/FHEW








```
    cout <<"Val1="<< val1 << " Binary: "<< bin1[3] << bin1[2] << bin1[1] << bin1[0] << endl;
    cout <<"Val2="<< val2 << " Binary: "<< bin2[3] << bin2[2] << bin2[1] << bin2[0] << endl;

    auto bin1_0 = cc.Encrypt(sk, bin1[0]);
     auto bin1_1 = cc.Encrypt(sk, bin1[1]);
   auto bin1_2 = cc.Encrypt(sk, bin1[2]);
    auto bin1_3 = cc.Encrypt(sk, bin1[3]);

        auto bin2_0 = cc.Encrypt(sk, bin2[0]);
     auto bin2_1 = cc.Encrypt(sk, bin2[1]);
   auto bin2_2 = cc.Encrypt(sk, bin2[2]);
    auto bin2_3 = cc.Encrypt(sk, bin2[3]);

        auto c_carryin = cc.Encrypt(sk, 0);


lWECiphertext c_sum1,c_carryout,c_sum2,c_sum3,c_sum4;

    tie(c_sum1,c_carryout)=FA(cc,bin1_0,bin2_0,c_carryin );
    tie(c_sum2,c_carryout)=FA(cc,bin1_1,bin2_1,c_carryout );
    tie(c_sum3,c_carryout)=FA(cc,bin1_2,bin2_2,c_carryout );
    tie(c_sum4,c_carryout)=FA(cc,bin1_3,bin2_3,c_carryout );
```

https://asecuritysite.com/openfhe/openfhe_11cpp
https://asecuritysite.com/openfhe/openfhe_11cpp_pke

MD/FHEW - MUX and Majority

a	b	c	Z
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

a	b	c	Z
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	1

```
std::cout << "Creating bootstrapping keys..." << std::endl;

cc.BTKeyGen(sk);

std::cout << "Completed key generation." << std::endl;

auto a = cc.Encrypt(sk, abit);
auto b = cc.Encrypt(sk, bbit);
auto c = cc.Encrypt(sk, cbit);

std::vector<LWECiphertext> bits;
bits.push_back(a);
bits.push_back(b);
bits.push_back(c);

auto ctMaj = cc.EvalBinGate(MAJORITY, bits);
auto cMux = cc.EvalBinGate(CMUX, bits);

LWEPlaintext resultMaj,resultMux;

cc.Decrypt(sk, ctMaj, &resultMaj);
cc.Decrypt(sk, cMux, &resultMux);

cout << "\na= " << abit << ", b= " << bbit << ", c= " << cbit << endl;
cout << "\nMajority: " << resultMaj << endl;
cout << "MUX: " << resultMux << endl;
```

https://asecuritysite.com/openfhe/openfhe_11cpp
https://asecuritysite.com/openfhe/openfhe_11cpp_pke

Chebyshev Functions

With approximation theory, it is possible to determine an approximate polynomial p(x)
that is approximate to a function f(x).

$$
\begin{aligned}
T_0(x) &= 1 \\
T_1(x) &= x \\
T_2(x) &= 2x^2 - 1 \\
T_3(x) &= 4x^3 - 3x \\
T_4(x) &= 8x^4 - 8x^2 + 1 \\
T_5(x) &= 16x^5 - 20x^3 + 5x \\
T_6(x) &= 32x^6 - 48x^4 + 18x^2 - 1 \\
T_7(x) &= 64x^7 - 112x^5 + 56x^3 - 7x \\
T_8(x) &= 128x^8 - 256x^6 + 160x^4 - 32x^2 + 1 \\
T_9(x) &= 256x^9 - 576x^7 + 432x^5 - 120x^3 + 9x \\
T_{10}(x) &= 512x^{10} - 1280x^8 + 1120x^6 - 400x^4 + 50x^2 - 1 \\
T_{11}(x) &= 1024x^{11} - 2816x^9 + 2816x^7 - 1232x^5 + 220x^3 - 11x
\end{aligned}
$$

```
 if (opt==0) {
     result = cc->EvalChebyshevFunction([](double x) -> double { return std::log10(x); }, cipherte
     std::cout <<" x    log10(x)\n----------" << std::endl;
 }
else if (opt==1) {
 result = cc->EvalChebyshevFunction([](double x) -> double { return std::log2(x); }, ciphertext, l
  std::cout <<" x    log2(x)\n----------" << std::endl;
}
else if (opt==2) {
 result = cc->EvalChebyshevFunction([](double x) -> double { return std::log(x); }, ciphertext, lo
    std::cout <<" x    ln(x)\n----------" << std::endl;
}
   else if (opt==3) {
     result = cc->EvalChebyshevFunction([](double x) -> double { return std::exp(x); }, ciphertext
    std::cout <<" x    exp(x)\n----------" << std::endl;
}
else if (opt==4) {
     result = cc->EvalChebyshevFunction([](double x) -> double { return std::exp2(x); }, ciphertex
    std::cout <<" x    2^x\n----------" << std::endl;
}
```

https://asecuritysite.com/openfhe/openfhe_18cpp

Polynomial Evaluation

A polynomial takes the form form of $p(x) = a_n.x^n + a_{n-1}.x^{n-1} + a_1.x + a_0$, and where $a_0 \ldots a_n$ are the coeffients of the powers, and n is the maximum power of the polynomial. With CKKS in OpenFHE, we can evaluate the result of a polynomial for a given range of x values. For example, if we have $p(x) = 5.x^2 + 3.x + 7$ will give a result of $p(2) = 33$.

```
CryptoContext<DCRTPoly> cc = GenCryptoContext(parameters);
cc->Enable(PKE);
cc->Enable(KEYSWITCH);
cc->Enable(LEVELEDSHE);
cc->Enable(ADVANCEDSHE);

size_t encodedLength = input.size();

Plaintext plaintext1 = cc->MakeCKKSPackedPlaintext(input);

auto keyPair = cc->KeyGen();

std::cout << "Generating evaluation key.";
cc->EvalMultKeyGen(keyPair.secretKey);

auto ciphertext1 = cc->Encrypt(keyPair.publicKey, plaintext1);

auto result = cc->EvalPoly(ciphertext1, coefficients1);


Plaintext plaintextDec;

cc->Decrypt(keyPair.secretKey, result, &plaintextDec);

plaintextDec->SetLength(encodedLength);
```

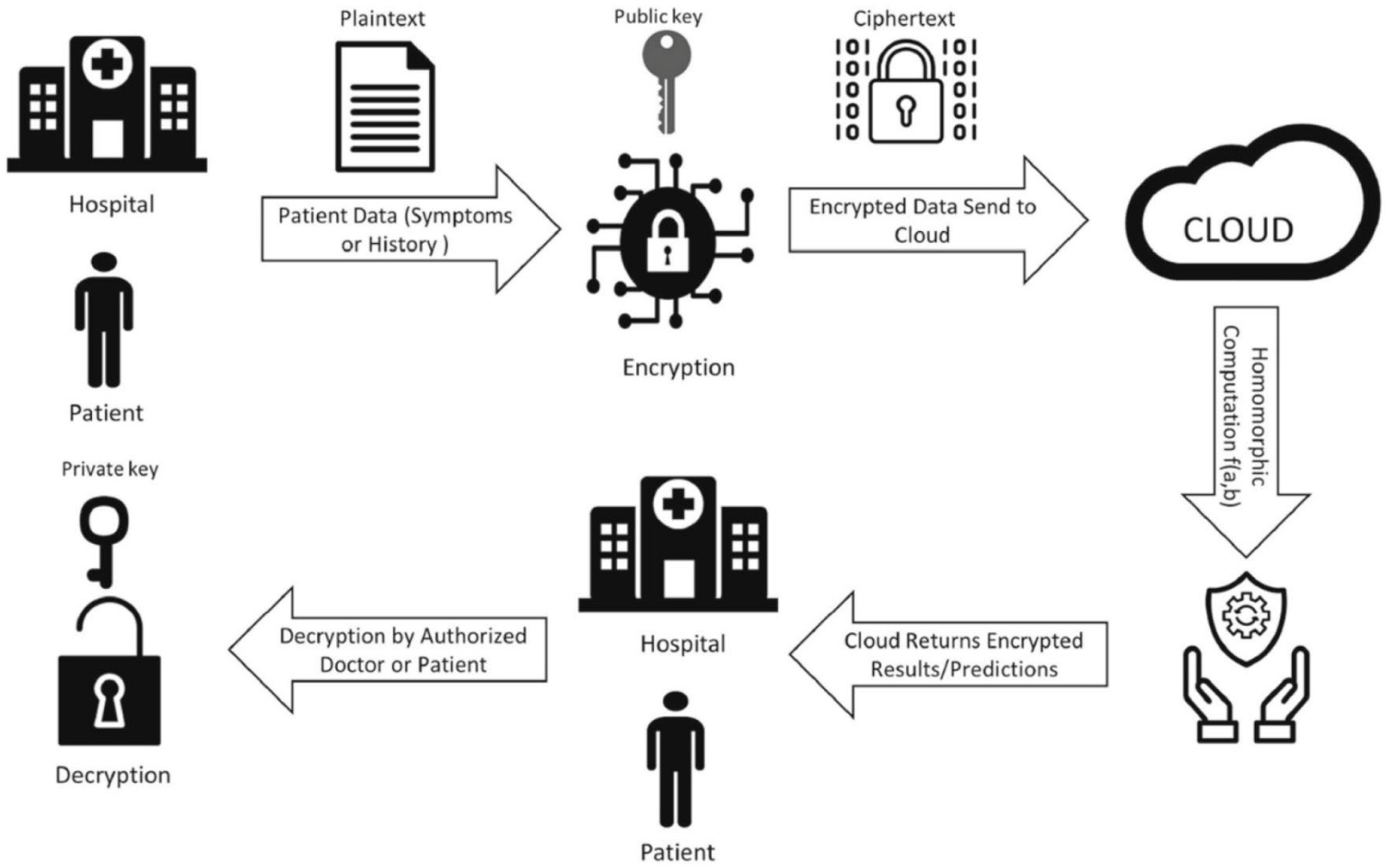
https://asecuritysite.com/openfhe/openfhe_20cpp

FHE:
4. Data Sharing

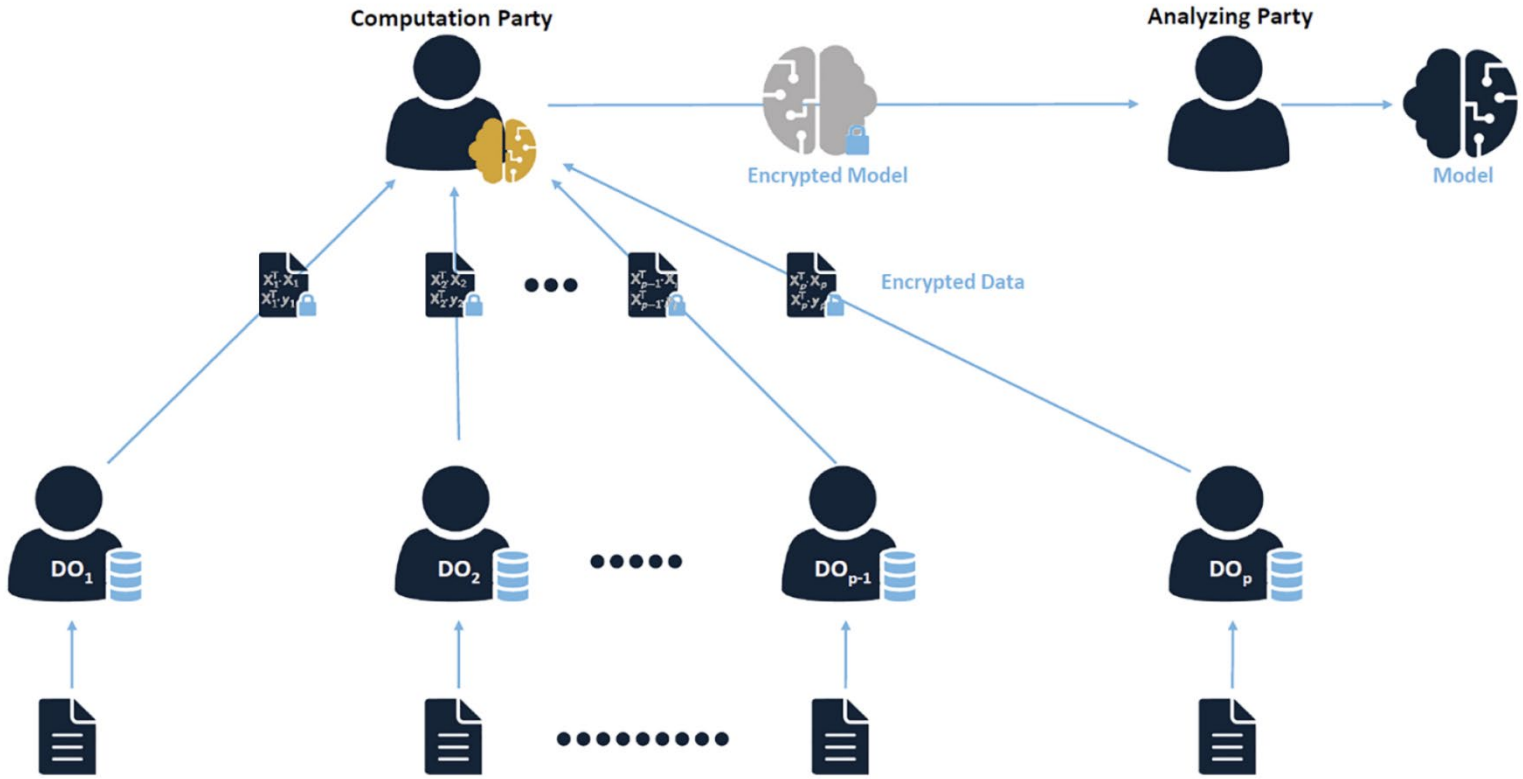
Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

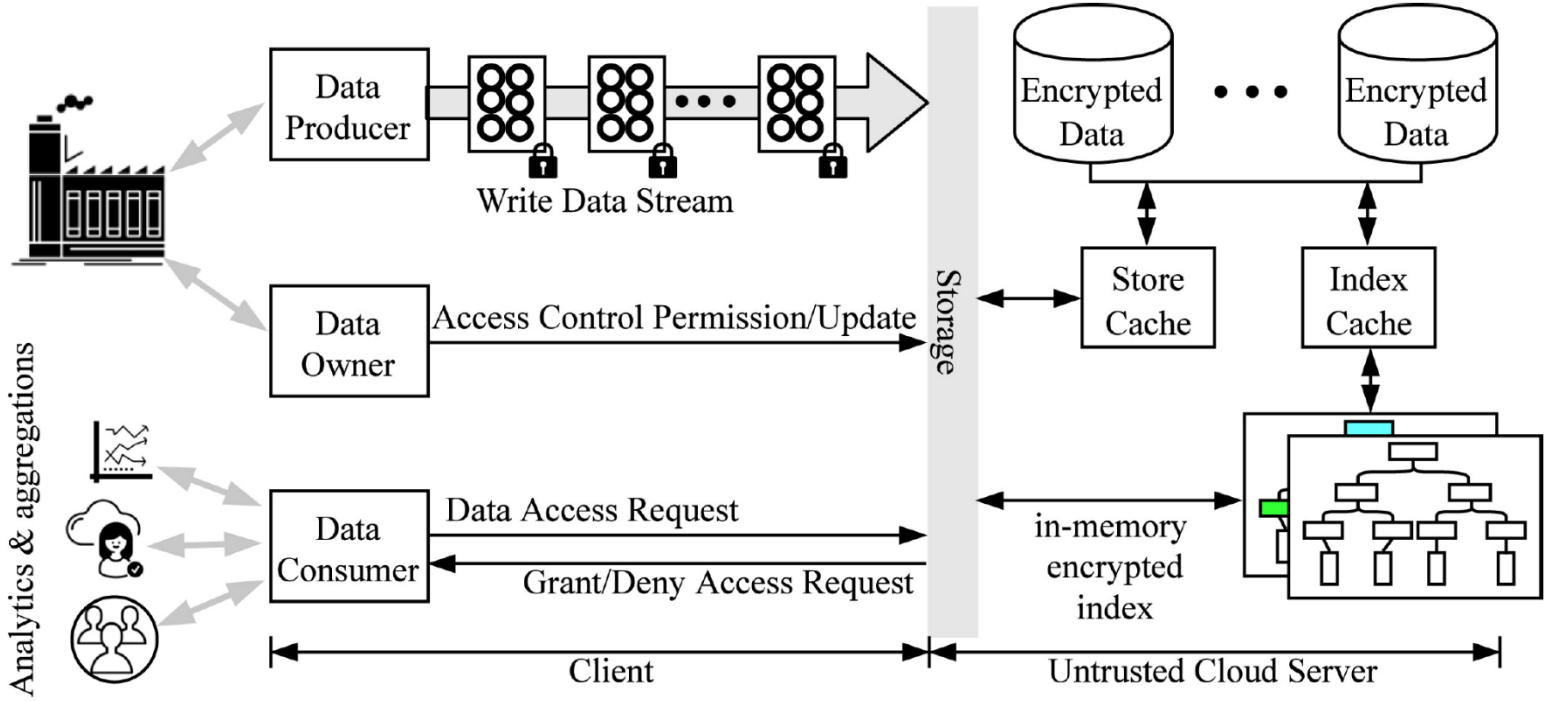
Homomorphic Sharing



Multiparty Sharing



SmartCrypt


Data Producer
Write Data Stream
Encrypted Data
Encrypted Data
Data Owner
Access Control Permission/Update
Storage
Store Cache
Index Cache
Analytics & aggregations
Data Consumer
Data Access Request
Grant/Deny Access Request
in-memory encrypted index
Client
Untrusted Cloud Server

Using Symmetric Key for HE

4.4.1 Order-preserving encryption (OPE)

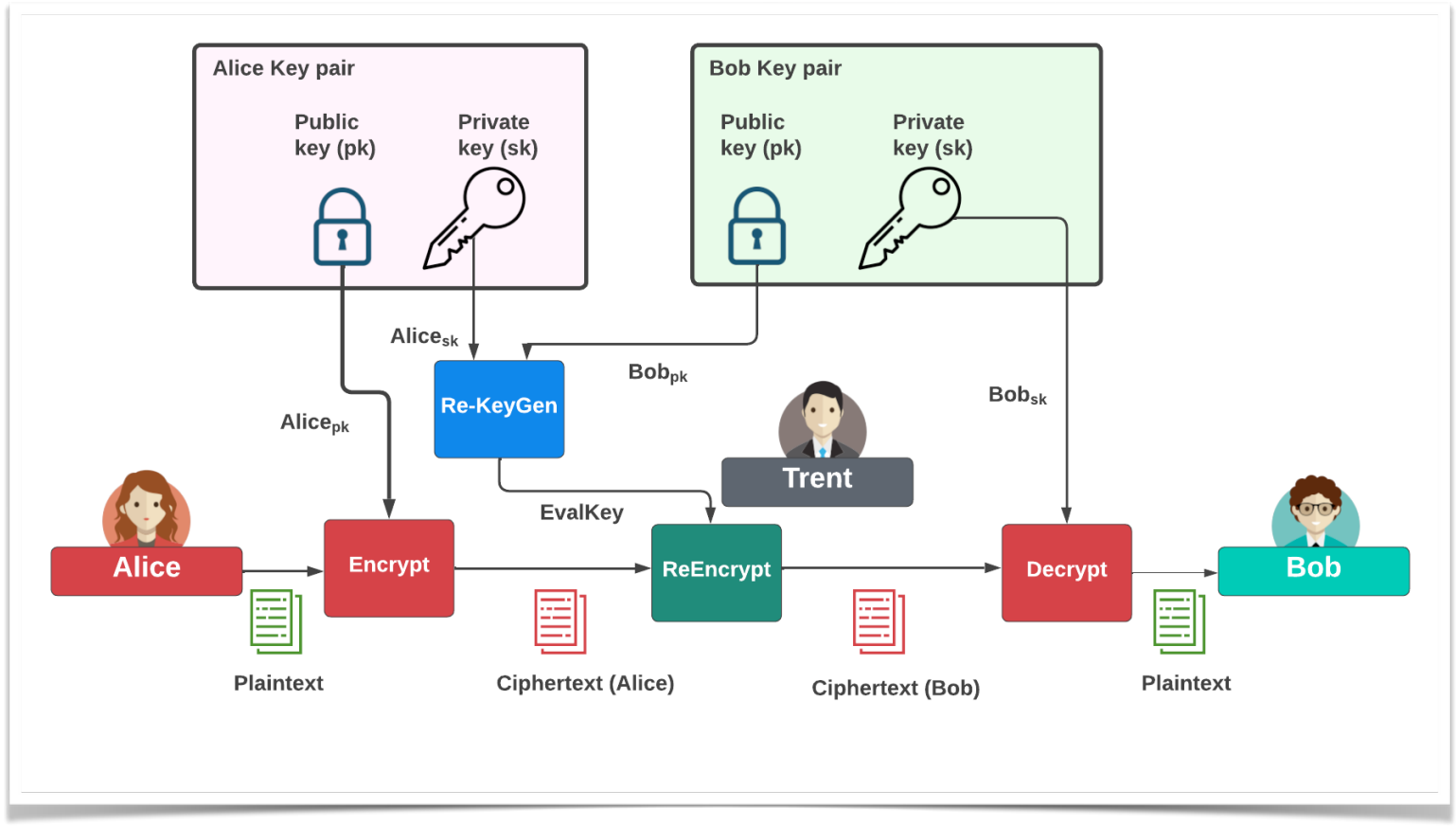
Rahman et al. outline an Order-preserving encryption (OPE) for encrypted searches using symmetric key encryption. [73]. Overall, OPE is a symmetric key encryption method that can be used to create an order on encrypted content. We will lose some information in supporting the ordering of the ciphers, but the security level should still stay acceptable for the application area. If we encrypt a value of A with a key of k to get $E_k(A)$, and then encrypt B to get $E_k(B)$. Then if $A < B$, then:

$$Ek(A) < Ek(B) \tag{4.1}$$

One of the best papers on the subject is [74], and an article is [75].

Proxy Re-encryption (PRE)

```
CCParams<CryptoContextCKKSRNS> parameters;
std::vector<double> dataInput = split(s1);


parameters.SetBatchSize(16);

parameters.SetMultiplicativeDepth(2);
parameters.SetScalingModSize(59);
parameters.SetSecurityLevel(SecurityLevel::HEStd_128_classic);

parameters.SetRingDim(16384);

parameters.SetPREMode(INDCPA);
parameters.SetKeySwitchTechnique(KeySwitchTechnique::HYBRID);

auto cc = GenCryptoContext(parameters);

cc->Enable(PKE);
cc->Enable(KEYSWITCH);
cc->Enable(LEVELEDSHE);
cc->Enable(PRE);
```



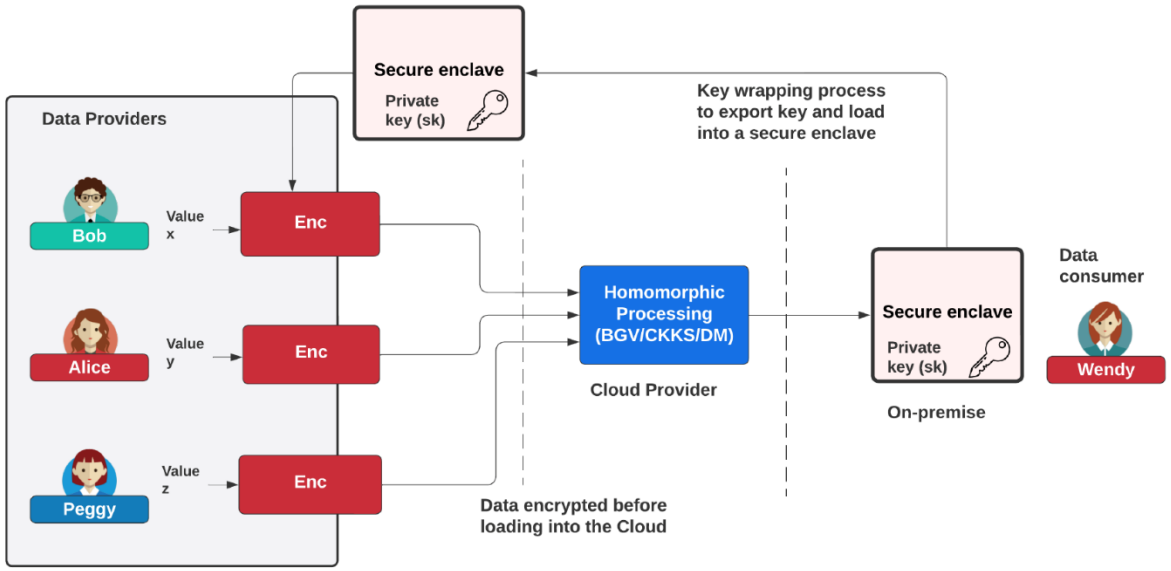
https://asecuritysite.com/openfhe/openfhe_21cpp

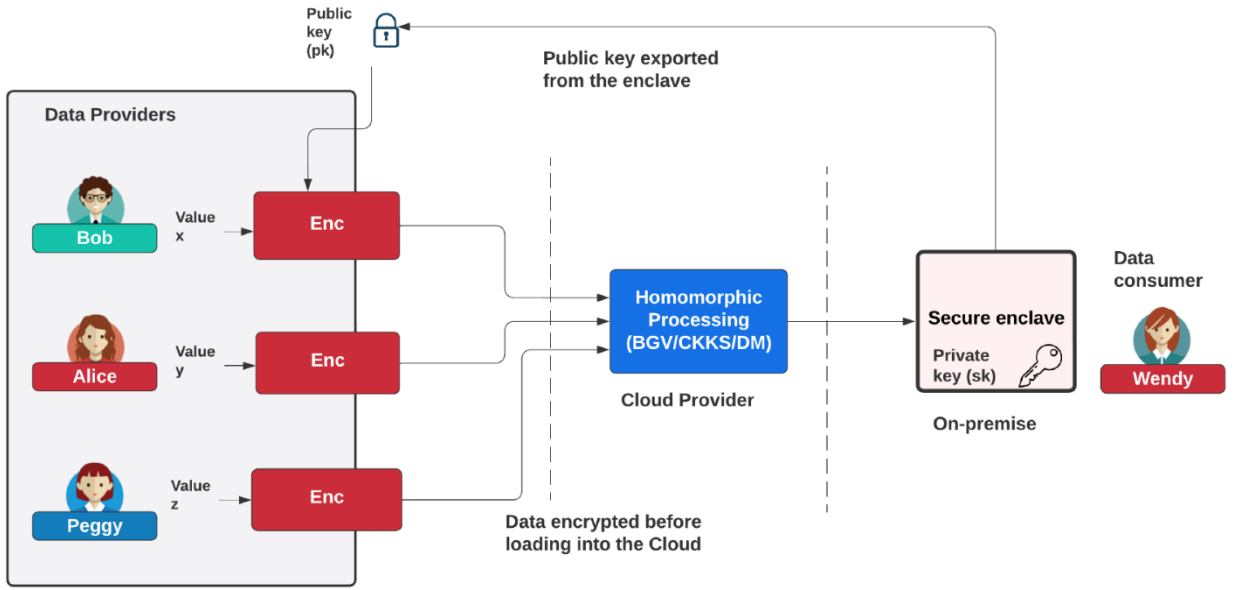
FHE:
5. Cloud

Prof Bill Buchanan OBE, Blockpass ID Lab

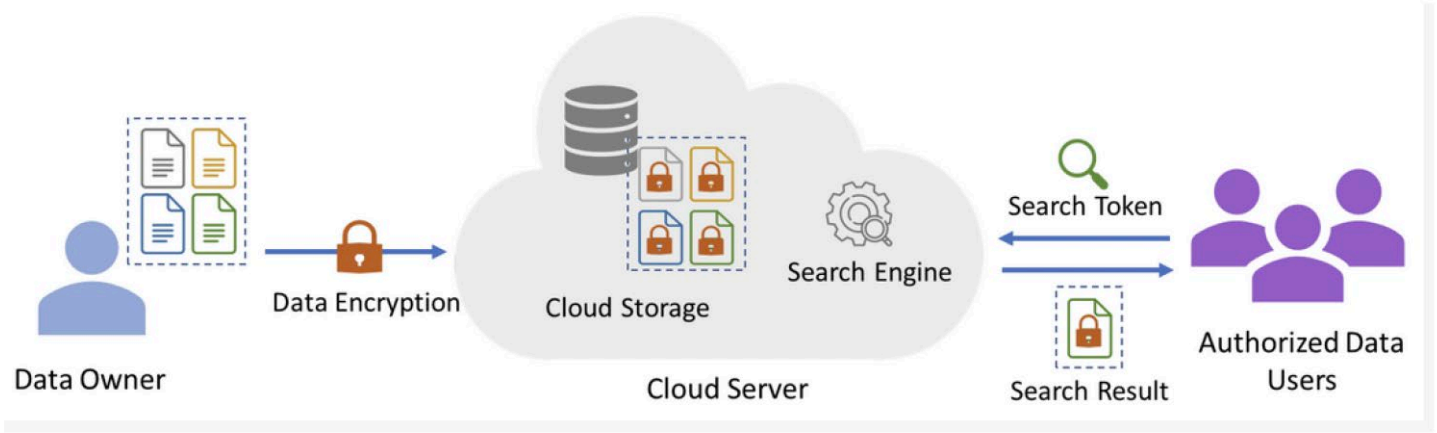
http://asecuritysite.com

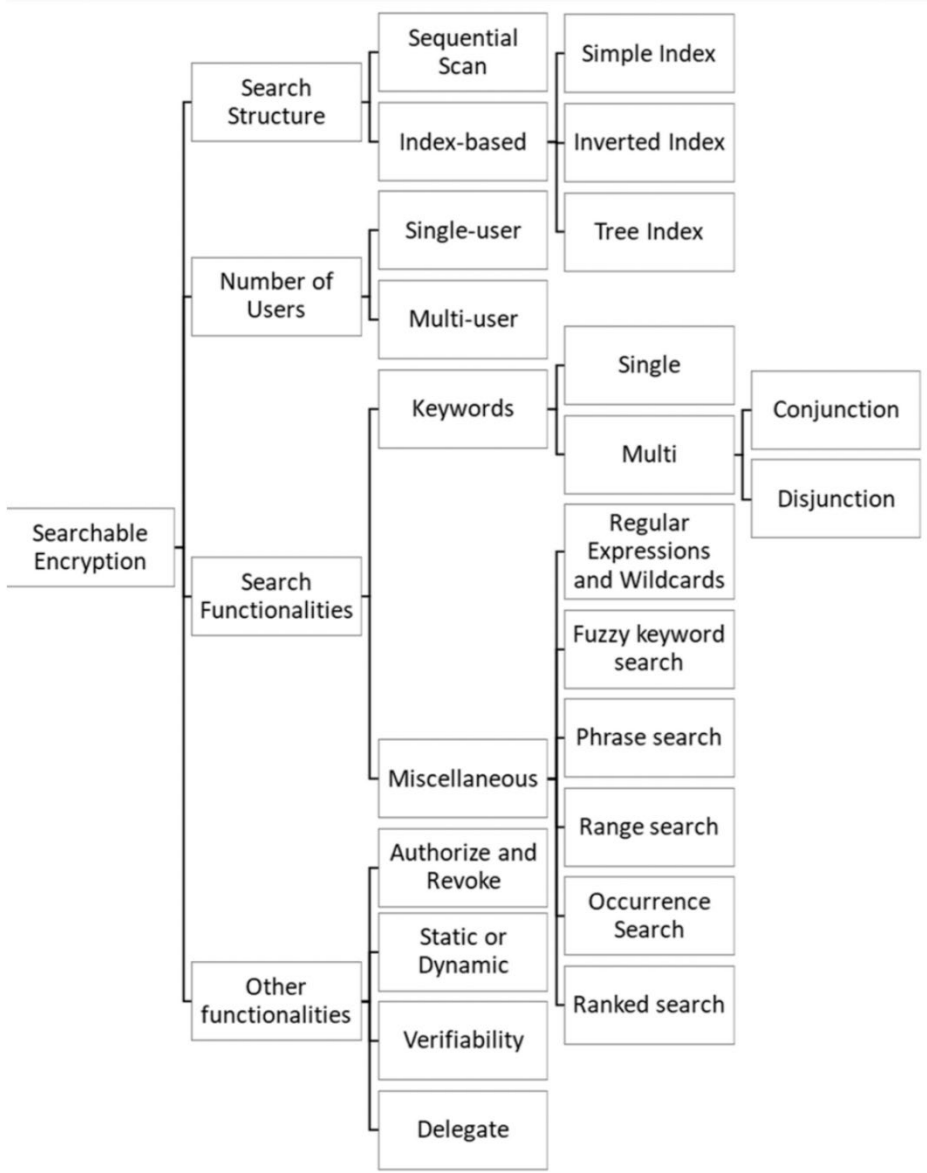
Symmetric/Asymmetric in Cloud





Searchable Encryption

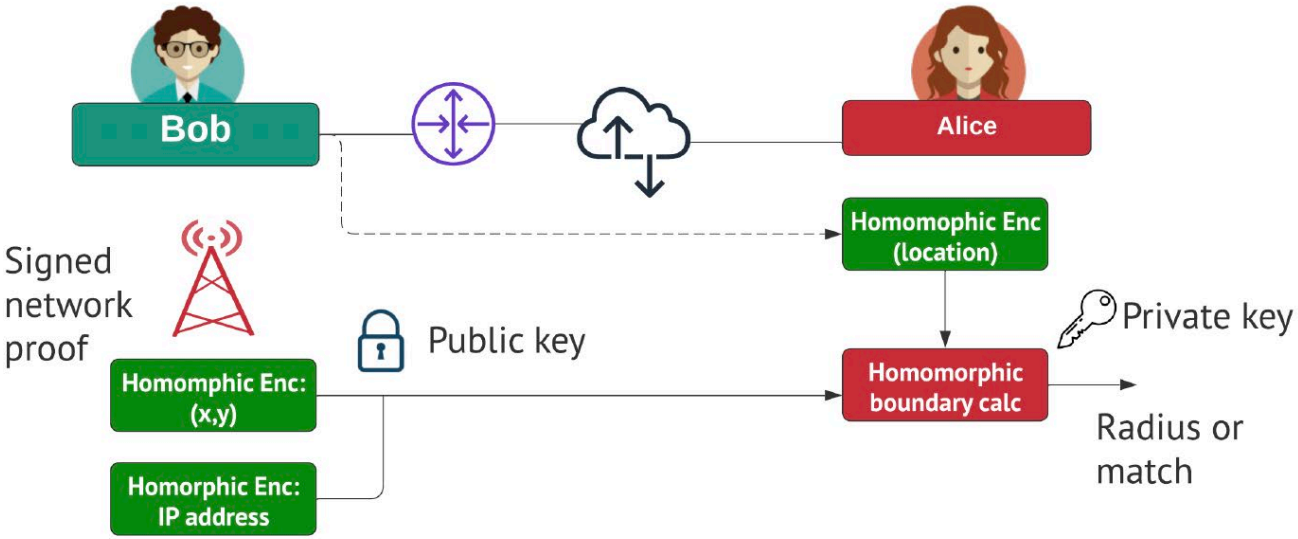




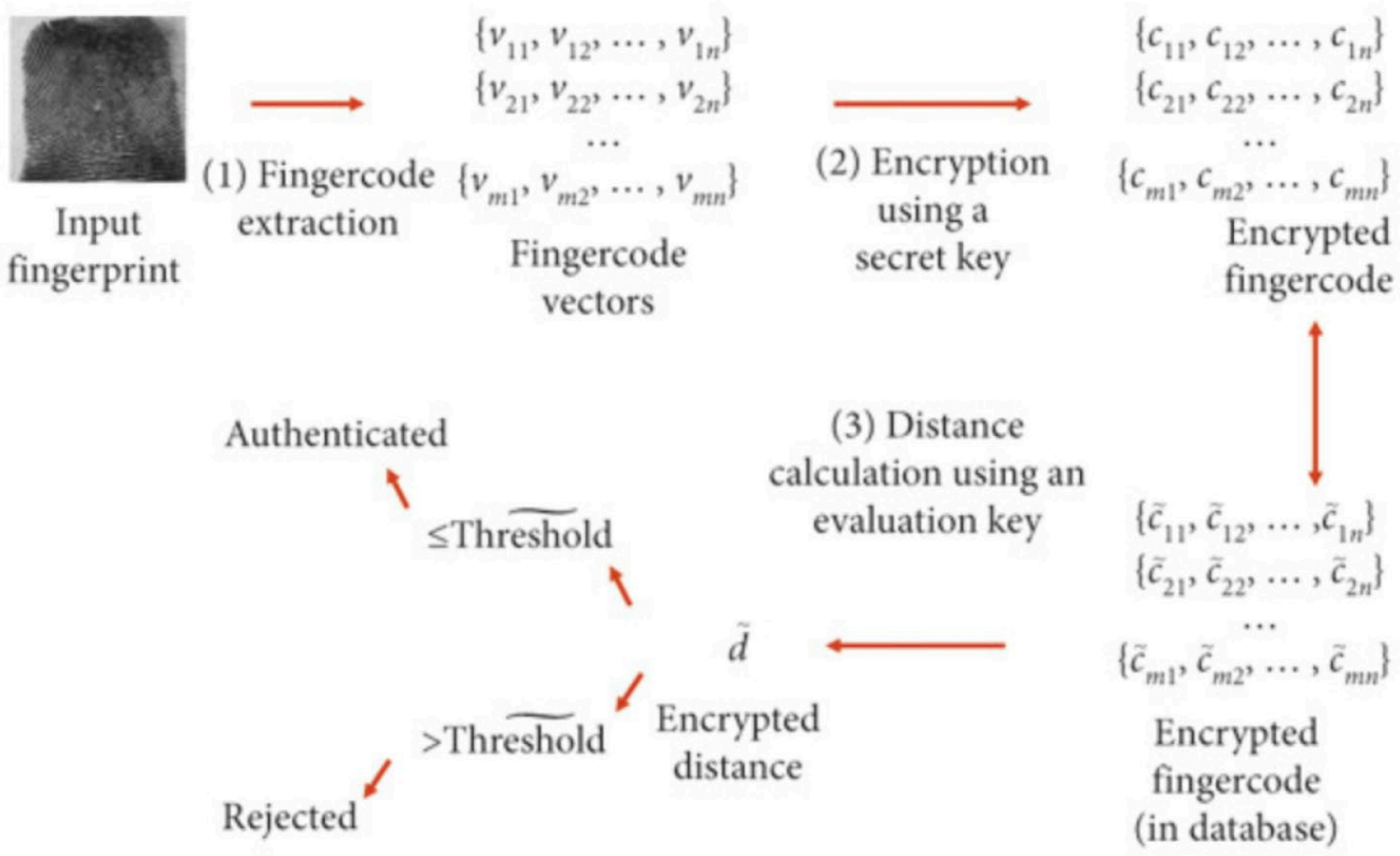
Location Tracking

```
E(TIME1)a E(Location1)a
E(TIME2)a E(Location2)a
E(TIME3)a E(Location2)a
E(TIME1)b E(Location1)b
E(TIME2)b E(Location2)b
E(TIME3)b E(Location2)b
```

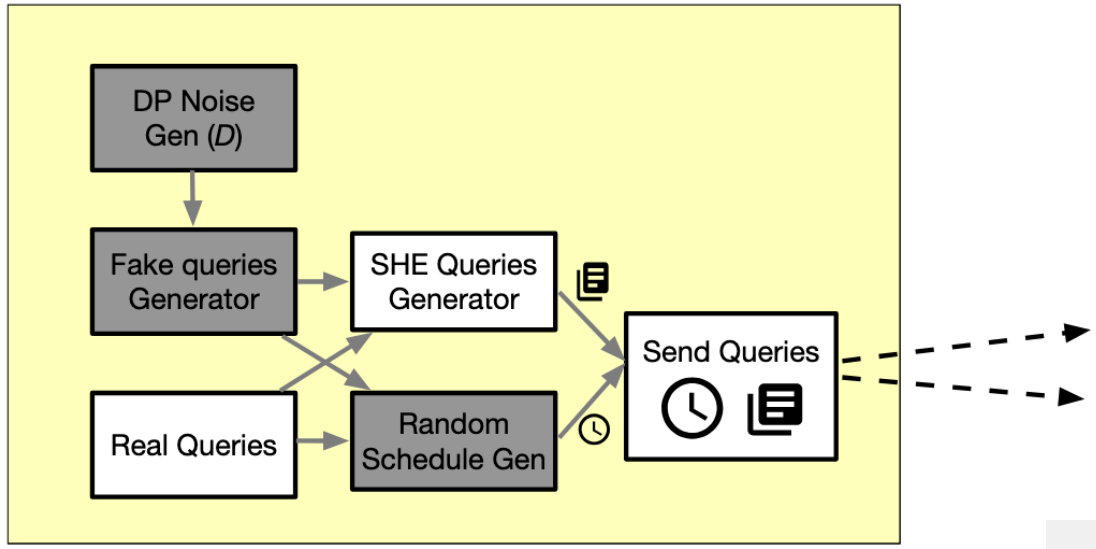
Timestamp	u_1 $rssi$	m	Type	u_2 $rssi$
1509240563.03	-64	D8:84:66:4C:D1:00	WiFi	-85
1509240563.03	-69	D8:84:66:4E:E4:F0	WiFi	-79
1509240563.03	-59	D8:84:66:4E:F0:04	BL	-91

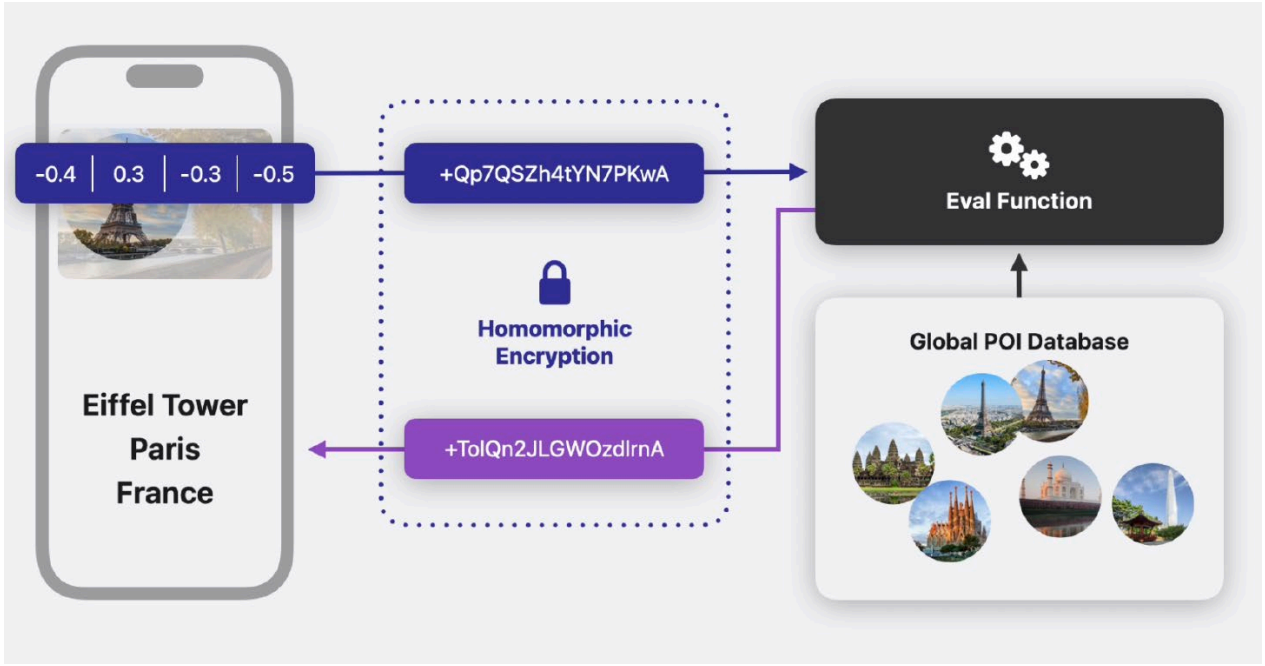


Privacy-aware biometrics/ID



Apple Privacy Aware





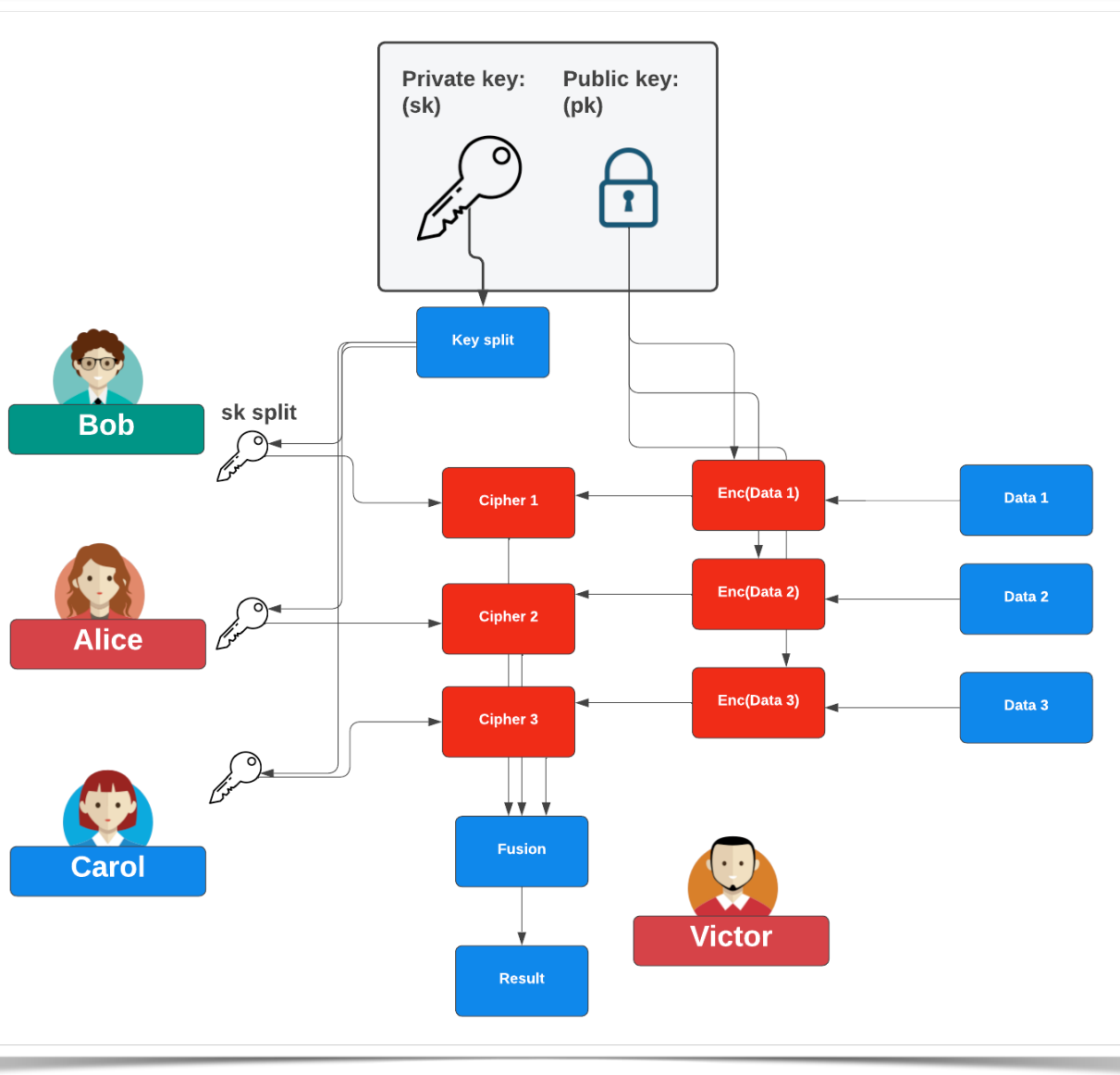
FHE:
6. MPC

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

Threshold Encryption

```
DCRTPoly partialPlaintext1;
DCRTPoly partialPlaintext2;
DCRTPoly partialPlaintext3;

Plaintext plaintextMultipartyNew;

const std::shared_ptr<CryptoParametersBase<DCRTPoly>> cryptoParams = bob.secretKey->GetCryptoPara
const std::shared_ptr<typename DCRTPoly::Params> elementParams     = cryptoParams->GetElementPara


auto ciphertextBob = cc->MultipartyDecryptLead({ ctAdd123}, bob.secretKey);
auto ciphertextAlice = cc->MultipartyDecryptMain({ ctAdd123}, alice.secretKey);

auto ciphertextCarol = cc->MultipartyDecryptMain({ ctAdd123}, carol.secretKey);

std::vector<Ciphertext<DCRTPoly>> partialCiphertextVec;
partialCiphertextVec.push_back(ciphertextBob[0]);
partialCiphertextVec.push_back(ciphertextAlice[0]);
partialCiphertextVec.push_back(ciphertextCarol[0]);

// partial decryptions are combined together
cc->MultipartyDecryptFusion(partialCiphertextVec, &plaintextMultipartyNew);
```



https://asecuritysite.com/openfhe/openfhe_16cpp
https://asecuritysite.com/openfhe/openfhe_17cpp

FHE:
7. ML

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

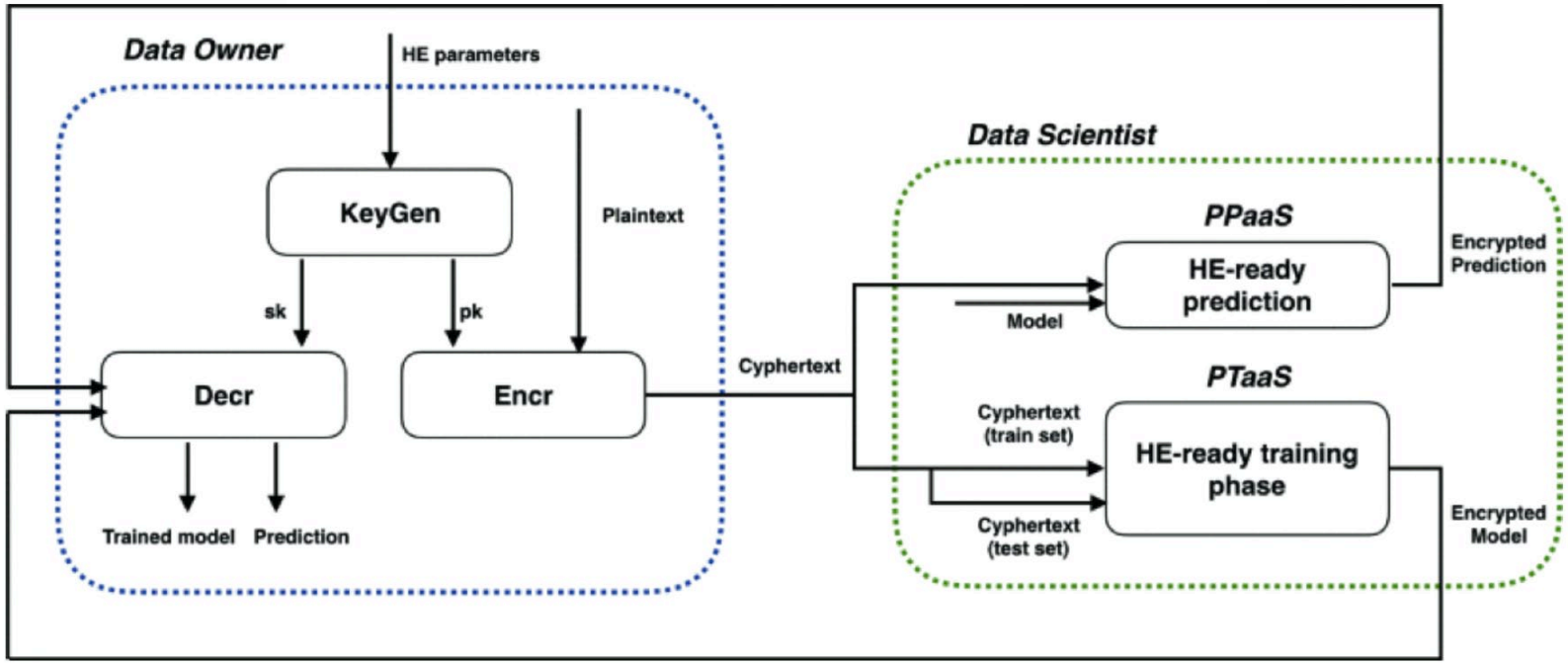
Matrix Operations

Iezzi et al. [87] define two methods of training with homomorphic encryption:

- Private Prediction as a Service (PPaaS). This is where the prediction is outsourced to a service provider who has a pre-trained model and where encrypted data is sent to the service provider. In this case, the data owner does not learn the model used.
- Private Training as a Service (PTaaS). This is where the data owner provides data to a service provider and who will train the model. The service provider can then provide a prediction for encrypted data.

Wood et al [56] adds models of:

- Private outsourced computation. This involves moving computation into the cloud.
- Private prediction. This involves homomorphic data processed into the cloud, and not having access to the training model.
- Private training. This is where a cloud entity trains a model based on the client's data.



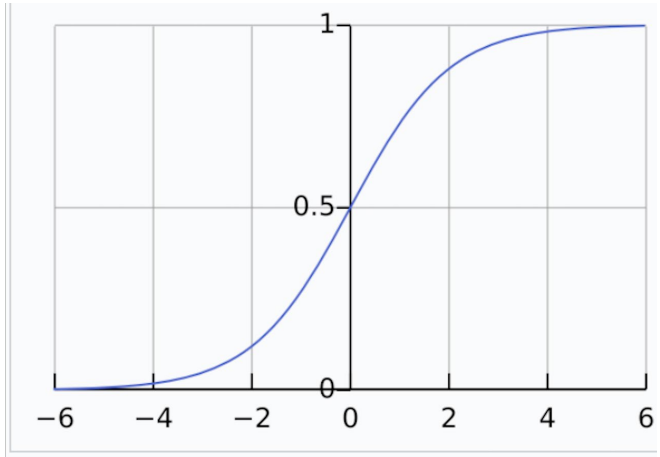
Private Prediction as a Service

Reference	PPaaS Model	HE scheme	Platform	Running Time	Dataset
Graepel et al. [54]	Linear Means Classifier	FV / Bra on SEAL	Intel Core i7 @2.8 GHz with 8GB of RAM	6 sec	Wisconsin Breast Cancer Data
	Fisher's Linear Discriminant classifier			20 sec	
Gilad-Bachrac et al. [32]	Prediction by NN with 5 layers	HE scheme based on [56] on SEAL	Intel Xeon E5-1620 @ 3.5GHz with 16GB RAM	570 sec for a prediction	MNIST
Costantino et al. [45]	Bag-of-word	BGV on HElib	Intel Core i7-6700 @ 3.40GHz with GB RAM	19 min	Tweets
Hesamifard et al. [34]	Prediction by Convolutional Neural Network with 6 layers	LHE on HElib	Intel Xeon E5-2640 @2.4GHz with 16GB RAM	320 sec	MNIST
				11686 sec	CIFAR-10
Masters et al. [31]	Nesterov's Accelerate GD-based logistic regression	CKKS on HElib	Titan V	4500 speed up on mult	Banco Bradesco financial transactions
Brutzkus et al. [33]	Same NN as [32] for prediction	BFV on SEAL	Azure standard VM with 8 vCPUs 32GB of RAM	0.29 sec for a prediction	MNIST
	Linear model trained with features generated by AlexNet			0.16 sec for prediction	CalTech-101
Al Badawi et al. [37]	Fasttext NN trained on plaintext data	CKKS on GPU	NVIDIA DGX-1 multi-GPU with 8 V100 cards	0.66 sec	AGNews
Podschwadt et al. [44]	Embedding layer + RNN layer with 128 units	CKKS on HElib	AMD Ryzen 5 2600 @ 3.5GHz with 32GB RAM.	547.6 sec for a batch of 128 samples	IMDb

[87] M. Iezzi, "Practical privacy-preserving data science with homomorphic encryption: an overview," in 2020 IEEE International Conference on Big Data (Big Data). IEEE, 2020, pp. 3979–3988.

Logistic Function

With homomorphic encryption we can represent a mathematical operation in the form for a homomorphic equation. One of the most widely used methods is to use Chebyshev polynomials, and which allows the mapping of the function to a Chebyshev approximation. In this case, we will use homomorphic encryption to approximate a logistic function (and which is represented by f(x)=1/(1+e^{-x}).



Standard logistic function where $L = 1, k = 1, x_0 = 0$.

$$f(x) = \frac{1}{1+e^{-x}}$$

```
std::cout << "Logistic Evaluation \n" << std::endl;

CCParams<CryptoContextCKKSRNS> parameters;
parameters.SetMultiplicativeDepth(5);
parameters.SetScalingModSize(40);

CryptoContext<DCRTPoly> cc = GenCryptoContext(parameters);
cc->Enable(PKE);
cc->Enable(KEYSWITCH);
cc->Enable(LEVELEDSHE);
cc->Enable(ADVANCEDSHE);

size_t encodedLength = input.size();

Plaintext plaintext1 = cc->MakeCKKSPackedPlaintext(input);

auto keyPair = cc->KeyGen();

std::cout << "Generating evaluation key.";
cc->EvalMultKeyGen(keyPair.secretKey);

auto ciphertext1 = cc->Encrypt(keyPair.publicKey, plaintext1);

auto result = cc->EvalLogistic(ciphertext1,-1,1,3);

Plaintext plaintextDec;

cc->Decrypt(keyPair.secretKey, result, &plaintextDec);

plaintextDec->SetLength(encodedLength);
```

https://asecuritysite.com/openfhe/openfhe_19cpp

Matrix Operations

The inner product of two vectors of a and b is represented by $\langle a, b \rangle$. It is the dot product of two vectors, and represented as $\langle a, b \rangle = a.ycos(\theta)$, and where θ is the angle between the two vectors.

If we have a vector of x=(10,20,15), then the magniture will be:

$$a = \sqrt{10^2 + 20^2 + 15^2} = 26.93$$

If we have the same vector of b=(10,20,15), we will have the same magnitude. The inner product will then be:

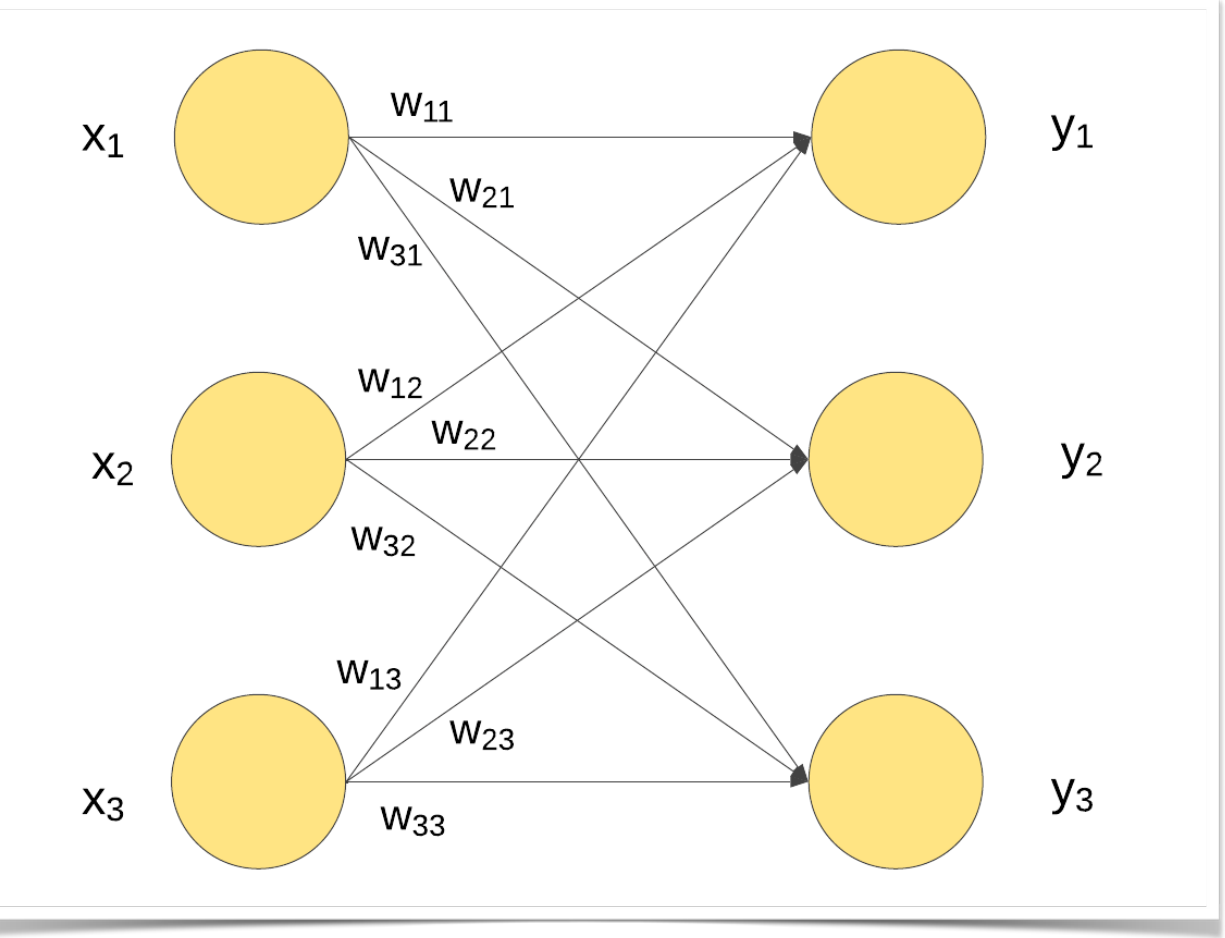
$$\langle a, b \rangle = |a|.|b|.cos(\theta) = 26.93 \times 26.93.cos(0) = 725$$

```
lbcrypto::CryptoContext<lbcrypto::DCRTPoly> cc;
cc = GenCryptoContext(parameters);

cc->Enable(PKE);
cc->Enable(LEVELEDSHE);
cc->Enable(ADVANCEDSHE);

KeyPair keys = cc->KeyGen();
cc->EvalMultKeyGen(keys.secretKey);
cc->EvalSumKeyGen(keys.secretKey);

Plaintext plaintext1 = cc->MakeCKKSPackedPlaintext(v1);
auto ct1             = cc->Encrypt(keys.publicKey, plaintext1);

Plaintext plaintext2 = cc->MakeCKKSPackedPlaintext(v2);
auto ct2           = cc->Encrypt(keys.publicKey, plaintext2);

auto finalResult     = cc->EvalInnerProduct(ct1, ct2, batchSize);
lbcrypto::Plaintext res;
cc->Decrypt(keys.secretKey, finalResult, &res);
res->SetLength(v1.size());
auto final = res->GetCKKSPackedValue()[0].real();

    std::cout <<  "v1=" << s1 << std::endl;
        std::cout <<  "v2=" << s2 << std::endl;
std::cout <<  "Inner Product Result: " << final << std::endl;
std::cout << "Expected value: " << inner_product(v1.begin(), v1.end(), v2.begin(), 0) << std::end
```

https://asecuritysite.com/openfhe/openfhe_13cpp

https://asecuritysite.com/openfhe/openfhe_14cpp

Matrix Operations



If we have a vector of the form:

$$v_1 = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

and a matrix of:

$$m_1 = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix}$$

We now get:

$$v_1 . m_1 = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix}$$

and:

$$v_1 . m_1 = \begin{bmatrix} x_1 . w_{11} + x_2 . w_{21} + x_3 . w_{31} & x_1 . w_{12} + x_2 . w_{22} + x_3 . w_{32} & x_1 . w_{13} + x_2 . w_{23} + x_3 . w_{33} \end{bmatrix}$$

Thus we get:

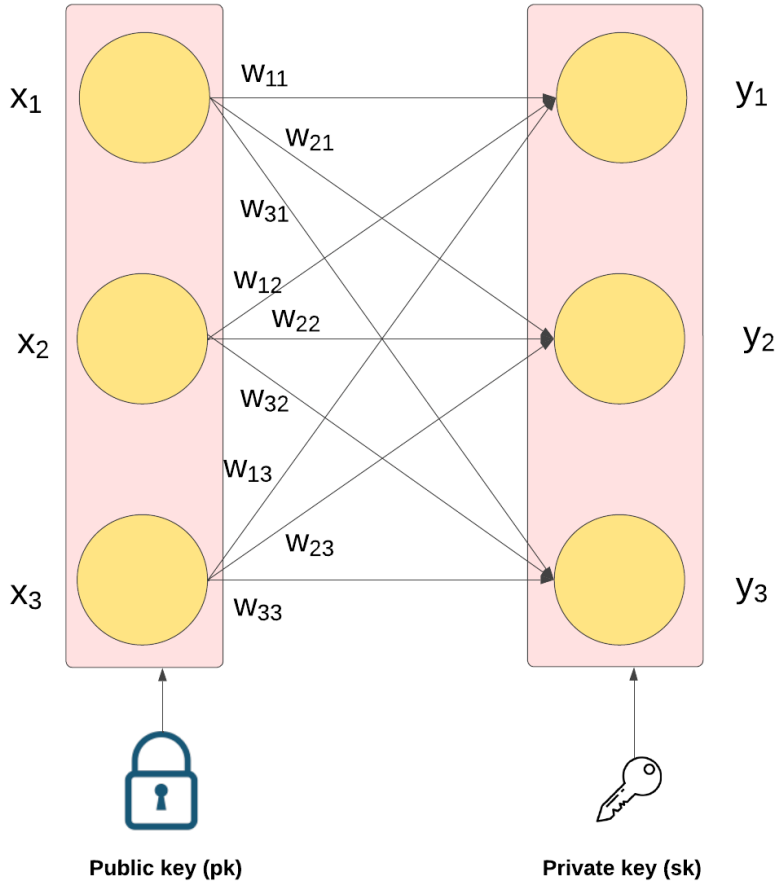
$$y_1 = x_1 . w_{11} + x_2 . w_{21} + x_3 . w_{31}$$

$$y_2 = x_1 . w_{12} + x_2 . w_{22} + x_3 . w_{32}$$

$$y_3 = x_1 . w_{13} + x_2 . w_{23} + x_3 . w_{33}$$

https://asecuritysite.com/openfhe/openfhe_27cpp

Matrix Operations

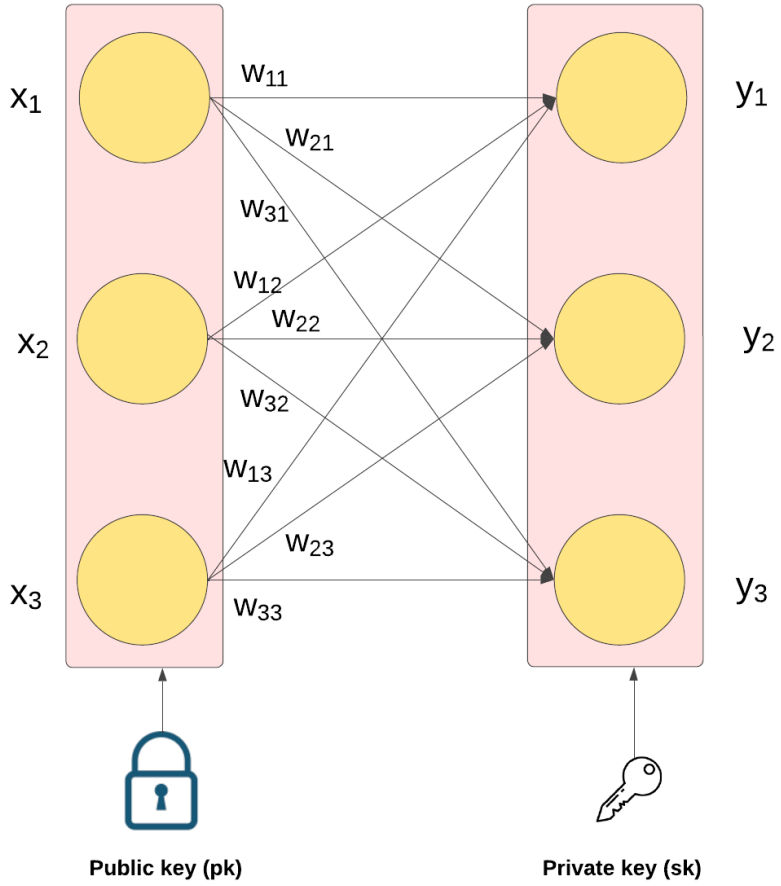



```
std::cout << "Generating rotation keys... ";
std::vector<int32_t> rotationKeys = {};
for (int i = -ROWS*COLS; i <= ROWS*COLS; i++) rotationKeys.push_back(i);
cryptoContext->EvalRotateKeyGen(keyPair.secretKey, rotationKeys);
std::cout << "Done"<< std::endl << std::endl;

std::vector<double> vector = genRandVect(ROWS, max,0);
Plaintext vectorP  = cryptoContext->MakeCKKSPackedPlaintext(vector);

std::vector<std::vector<double>> matrix = genRandMatrix(ROWS, COLS, max,1);

std::cout << "Vector (V1) = " << vector << std::endl;
std::cout << "Matrix (M1) = " << matrix << std::endl;

Ciphertext<DCRTPoly> vectorC = cryptoContext->Encrypt(keyPair.publicKey, vectorP);

Ciphertext<DCRTPoly> resC;
Plaintext res;
std::vector<double> resOutput, resOutputtmp;

resOutput = vectorMatrixMult(vector, matrix);

std::cout << "V1*M1   = " << resOutput << std::endl;

resC = vectorMatrixMultByInnProdCP(cryptoContext, keyPair.publicKey, vectorC, matrix);

cryptoContext->Decrypt(keyPair.secretKey, resC, &res);
res->SetLength(COLS);

std::cout << "vectorC * matrix (by inner product)      = " << res->GetCKKSPackedValue()  << std::en
```

https://asecuritysite.com/openfhe/openfhe_27cpp

https://asecuritysite.com/openfhe/openfhe_26cpp

Matrix Operations

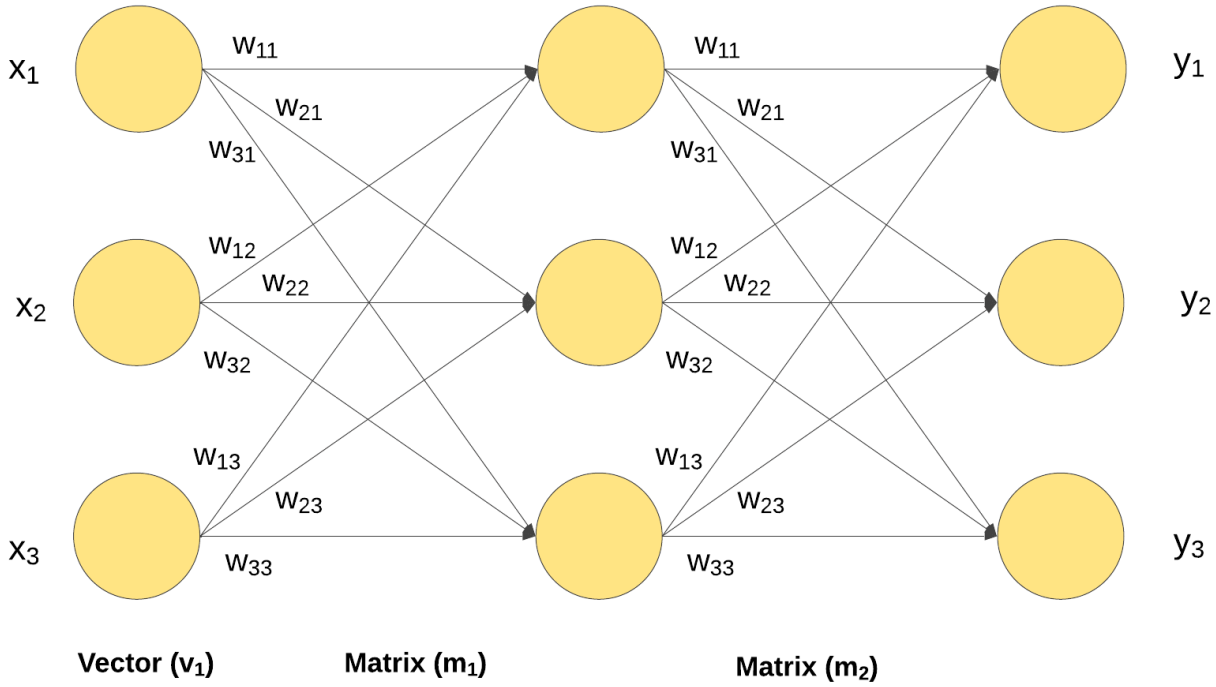



```
Ciphertext<DCRTPoly> vectorC = cryptoContext->Encrypt(keyPair.publicKey, vectorP);

Ciphertext<DCRTPoly> resC;
Plaintext res;
std::vector<int64_t> resOutput, resOutputtmp;


 resOutput = vectorMatrixMult(vector, matrix1);
resOutput = vectorMatrixMult(resOutput, matrix2);

std::cout << "V1*M1*M2 (non homomorphic)  = " << resOutput << std::endl;

//resC = vectorMatrixMultByInnProdCP(cryptoContext, keyPair.publicKey, vectorC, matrix1);
//resC = vectorMatrixMultByInnProdCP(cryptoContext, keyPair.publicKey, resC, matrix2);
resC = vectorMatrixMultByInnProdFastCP(cryptoContext, keyPair.publicKey, vectorC, matrix1);
 resC = vectorMatrixMultByInnProdFastCP(cryptoContext, keyPair.publicKey, resC, matrix2);

cryptoContext->Decrypt(keyPair.secretKey, resC, &res);
res->SetLength(n3);
resOutput = res->GetPackedValue();
std::cout << "V1*M1*M3 (by inner product)      = " << resOutput  << std::endl;
```

https://asecuritysite.com/openfhe/openfhe_27cpp

https://asecuritysite.com/openfhe/openfhe_26cpp

Matrix Operations

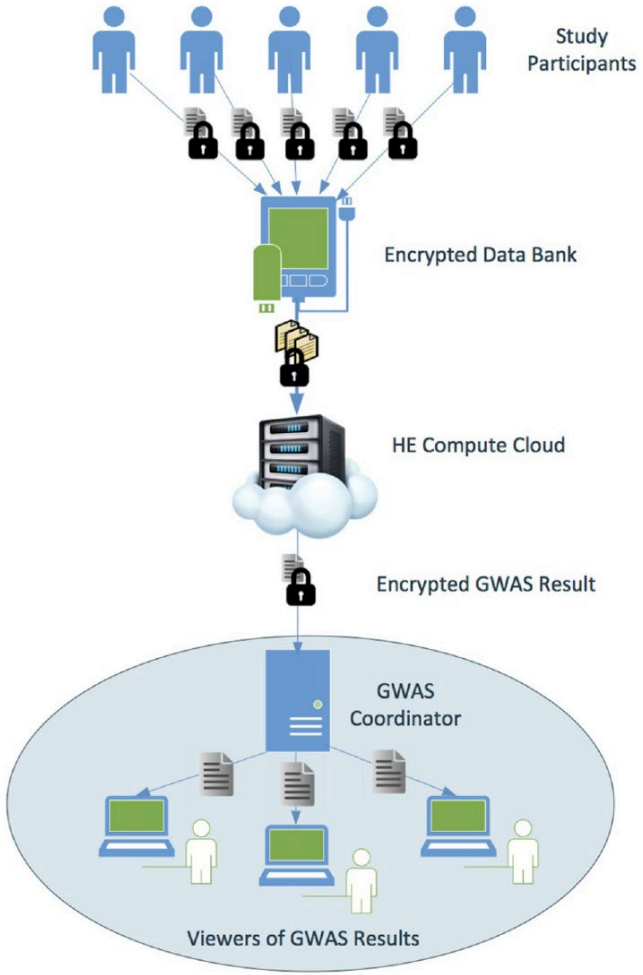



```
Ciphertext<DCRTPoly> vectorC = cryptoContext->Encrypt(keyPair.publicKey, vectorP);

Ciphertext<DCRTPoly> resC;
Plaintext res;
std::vector<int64_t> resOutput, resOutputtmp;


 resOutput = vectorMatrixMult(vector, matrix1);
resOutput = vectorMatrixMult(resOutput, matrix2);

std::cout << "V1*M1*M2 (non homomorphic)  = " << resOutput << std::endl;

//resC = vectorMatrixMultByInnProdCP(cryptoContext, keyPair.publicKey, vectorC, matrix1);
//resC = vectorMatrixMultByInnProdCP(cryptoContext, keyPair.publicKey, resC, matrix2);
resC = vectorMatrixMultByInnProdFastCP(cryptoContext, keyPair.publicKey, vectorC, matrix1);
 resC = vectorMatrixMultByInnProdFastCP(cryptoContext, keyPair.publicKey, resC, matrix2);

cryptoContext->Decrypt(keyPair.secretKey, resC, &res);
res->SetLength(n3);
resOutput = res->GetPackedValue();
std::cout << "V1*M1*M3 (by inner product)      = " << resOutput  << std::endl;
```

https://asecuritysite.com/openfhe/openfhe_28cpp

https://asecuritysite.com/openfhe/openfhe_29cpp

GWAS (Gnome-wide Association Studies)



	GLM		HE LRA		HE Chisq	
SNP	OR	stat	OR	stat	OR	stat
rs10033900_T	1.09	1.97	1.08	1.91	1.06	1.44
rs943080_C	0.88	−2.94	0.89	−2.88	0.91	−2.26
rs79037040_G	0.88	−2.98	0.88	−2.91	0.89	−2.82
rs2043085_T	0.91	−2.01	0.92	−1.95	0.92	−2.13
rs2230199_C	1.41	6.83	1.38	6.67	1.40	7.10
rs8135665_T	1.12	2.04	1.12	2.03	1.12	2.29
rs114203272_T	0.62	−3.55	0.63	−3.50	0.67	−3.08
rs114212178_T	0.87	−0.70	0.87	−0.69	0.86	−0.77

Figure 7.8: Results for GWAS [92]

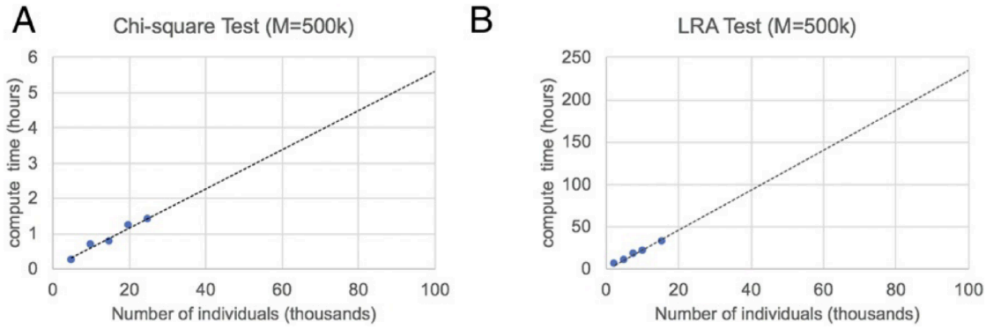
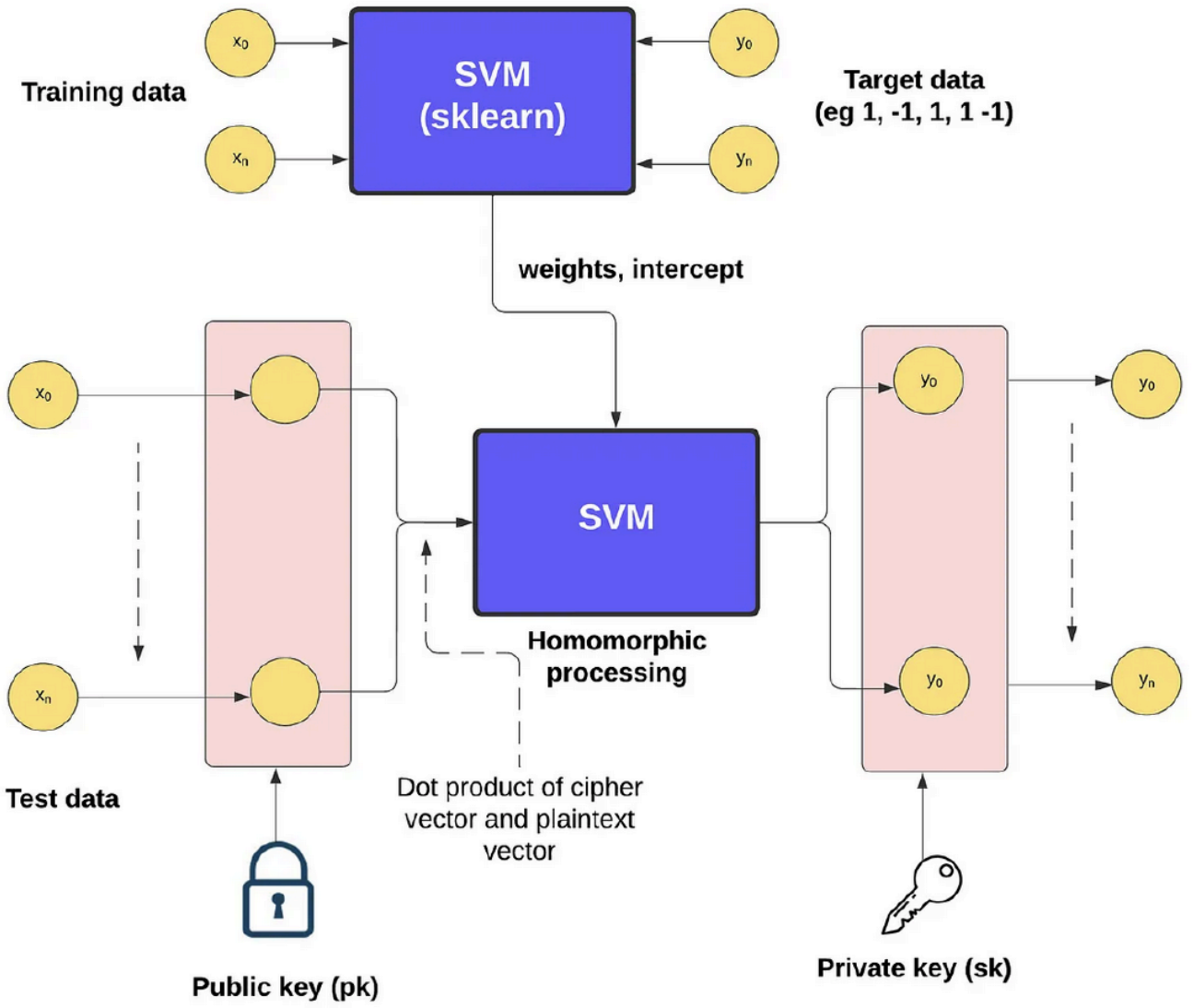


Figure 7.9: Results for GWAS [92]

SVM


Training data
SVM (sklearn)
Target data (eg 1, -1, 1, 1 -1)
weights, intercept
SVM
Homomorphic processing
Test data
Dot product of cipher vector and plaintext vector
Public key (pk)
Private key (sk)

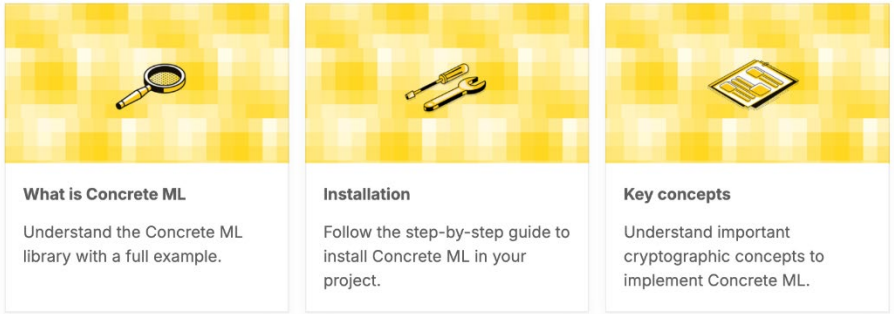
Concrete-ML

Welcome

Concrete ML is an open-source, privacy-preserving, machine learning framework based on Fully Homomorphic Encryption (FHE).

Get started

Learn the basics of Concrete ML, set it up, and make it run with ease.



What is Concrete ML

Understand the Concrete ML library with a full example.

Installation

Follow the step-by-step guide to install Concrete ML in your project.

Key concepts

Understand important cryptographic concepts to implement Concrete ML.

Concrete-ML also allows for a direct replacement for the following methods:

- Linear models: LinearRegression, LogisticRegression, LinearSVC, LinearSVR PoissonRegressor, TweedieRegressor, GammaRegressor, Lasso, Ridge, ElasticNet and SGDRegressor.

- Tree Models: DecisionTreeClassifier, DecisionTreeRegressor, RandomForestClassifier, and RandomForestRegressor.
- NeuralNetClassifier (MLPClassifier) and NeuralNetRegressor (MLPRegressor).
- Nearest Neighbour. KNeighborsClassifier

FHE:
8. Quantum Robust

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

Quantum Robust

- Existing state-of-the-art gives 105 Qubits (Google Willow). Need millions to crack existing public key methods.
- All existing public key methods will be cracked by QC, eg RSA, ECC and ElGamal.
- Kyber for Key Exchange/Encryption and Dilithium for Digital Signatures should be used.
- LWE is currently a quantum robust method.

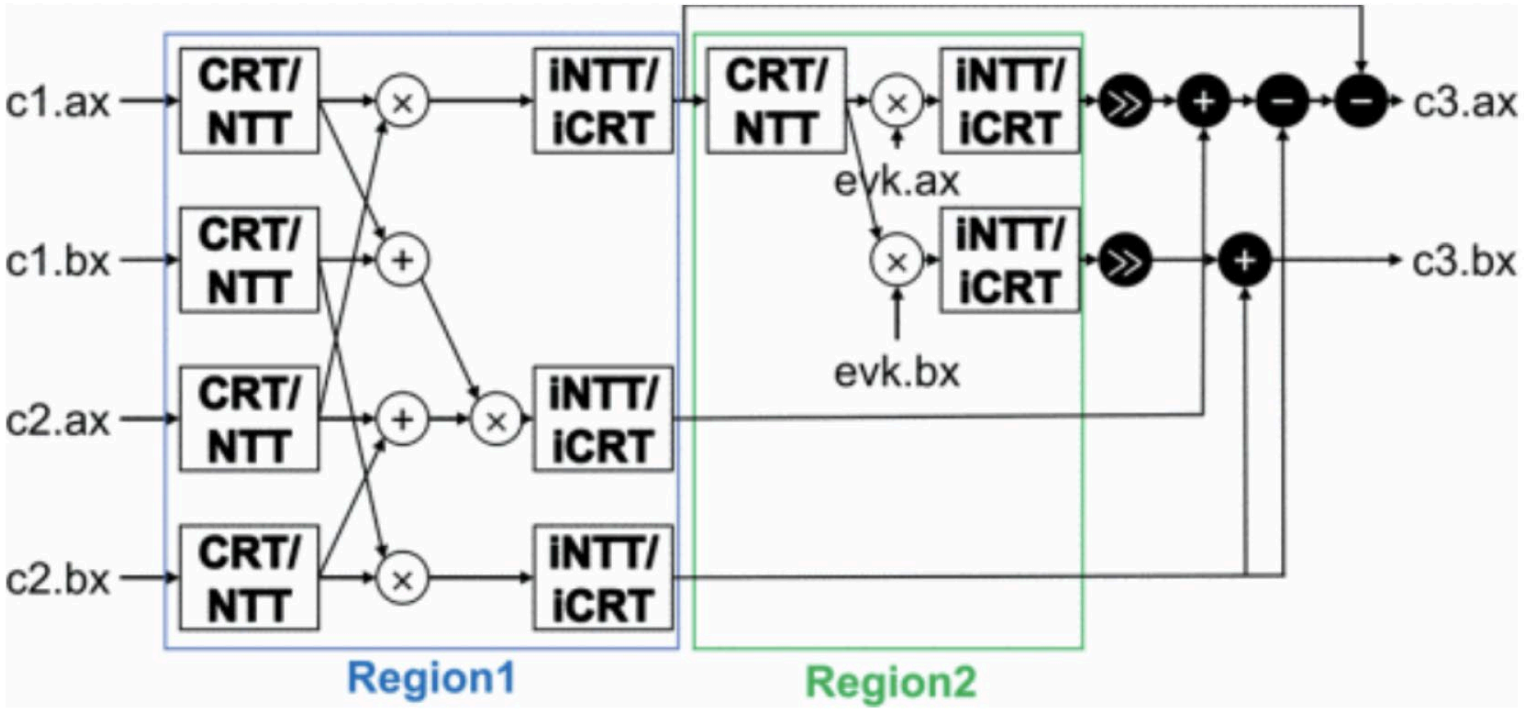
FHE:
9. Hardware Requirements

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

NTT (Number Theoretic Transform)

$$y_k = \sum_{j=0}^{n-1} x_j \alpha^{jk} \mod m$$

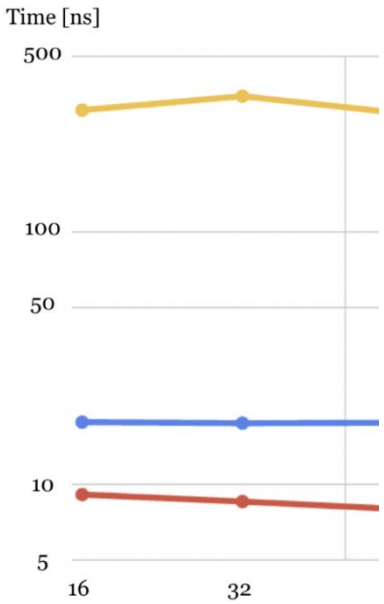


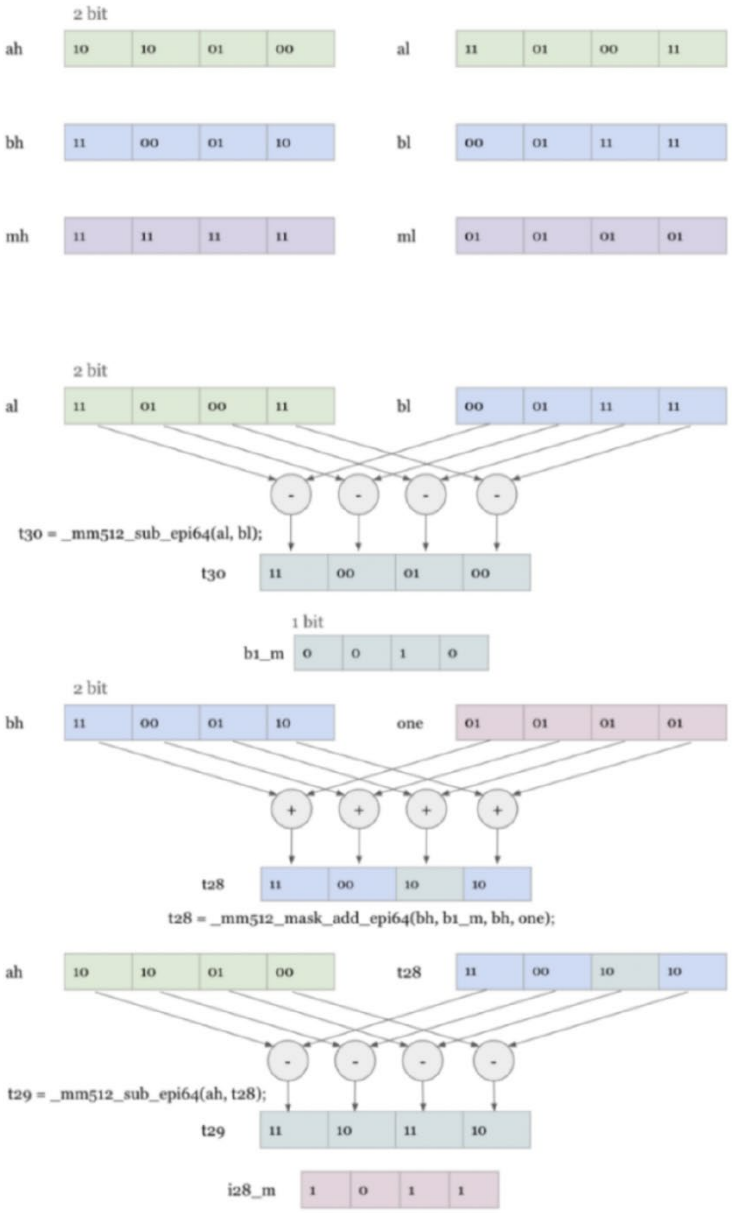
Two cipher text values: c1.ax and c2.bx

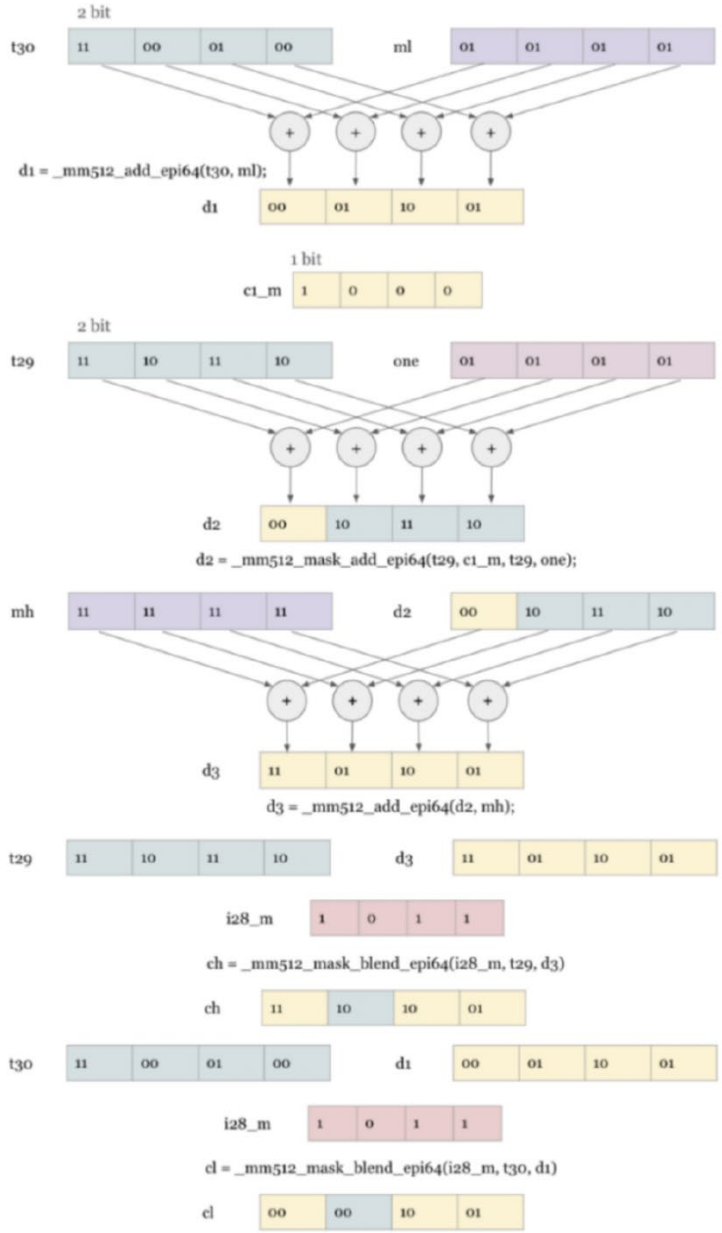
$$c3.ax = c1.axc2.ax, c3.cx = c1.bxc2.bx, c3.bx = c1.axc2.bx + c1.bxc2.ax$$

AVX-512

```
// Parallel modul
uint128_t submod1
    t30 = _mm512_
    c1_m = _mm512
    t28 = _mm512_
    t29 = _mm512_
    i28_m = _mm51
    ...
    d3 = _mm512_a
    *ch = _mm512_
    *cl = _mm512_
}
```







NTT Size

[Relative speedup]
AVX-MT-24
13.5 [46.6×]
7.8 [73.6×]
8.7 [66.6×]
44.1 [37.1×]
13.1 [12.3×]
87.3 [**41.1**×]

GPUs (BFV)

Operation	n	$\log_2 q$	GPU with [29] NTT RTX3060Ti	GPU with [29] NTT GTX1080	GPU with new NTT RTX3060Ti	GPU with new NTT GTX1080	[22] Tesla V100	SEAL CPU	T T_s
	2^{12}	109	4 μs	4.6 μs	4 μs	4.6 μs	-	14 μs	3.5×
...	2^{13}	218	5.1 μs	6.2 μs	5.1 μs	6.1 μs	-	58 μs	11.37×

	Instance Size	GPU	GPU Memory (GiB)	vCPUs	Memory (GiB)	Storage (GB)	Network Bandwidth (Gbps)	EBS Bandwidth (Gbps)	On Demand Price/hr*	1-yr ISP Effective Hourly (Linux)	3-yr ISP Effective Hourly (Linux)
Single GPU VMs	g5.xlarge	1	24	4	16	1x250	Up to 10	Up to 3.5	$1.006	$0.604	$0.402
	g5.2xlarge	1	24	8	32	1x450	Up to 10	Up to 3.5	$1.212	$0.727	$0.485
	g5.4xlarge	1	24	16	64	1x600	Up to 25	8	$1.624	$0.974	$0.650
	g5.8xlarge	1	24	32	128	1x900	25	16	$2.448	$1.469	$0.979
	g5.16xlarge	1	24	64	256	1x1900	25	16	$4.096	$2.458	$1.638
Multi GPU VMs	g5.12xlarge	4	96	48	192	1x3800	40	16	$5.672	$3.403	$2.269
	g5.24xlarge	4	96	96	384	1x3800	50	19	$8.144	$4.886	$3.258
	g5.48xlarge	8	192	192	768	2x3800	100	19	$16.288	$9.773	$6.515

	100	60.8 W	3763901 μs	44.61 W	71858 μs
	500	64.16 W	18786647 μs	48.60 W	337236 μs
2^{15}	1	59.55 W	186334 μs	44.29 W	3382 μs
	10	59.46 W	1796976 μs	45.48 W	36301 μs
	100	60.89 W	17822724 μs	55.71 W	338212 μs
	500	66.93 W	89749372 μs	65.13 W	342199 μs

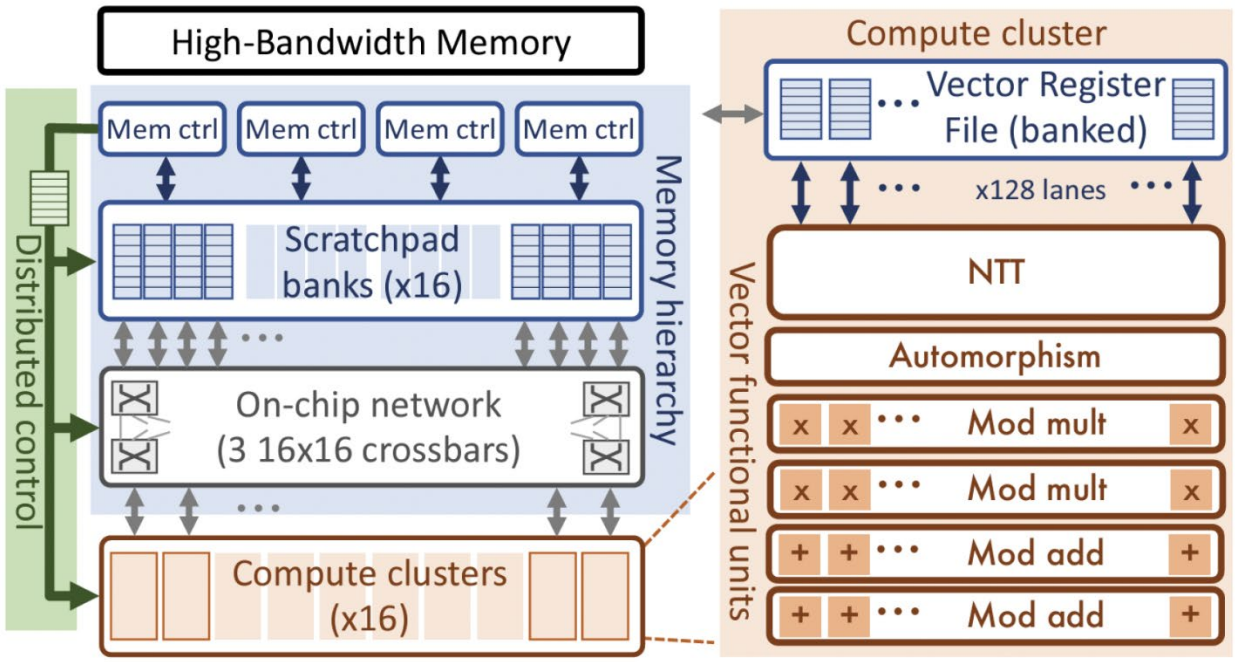
TX 3060Ti)

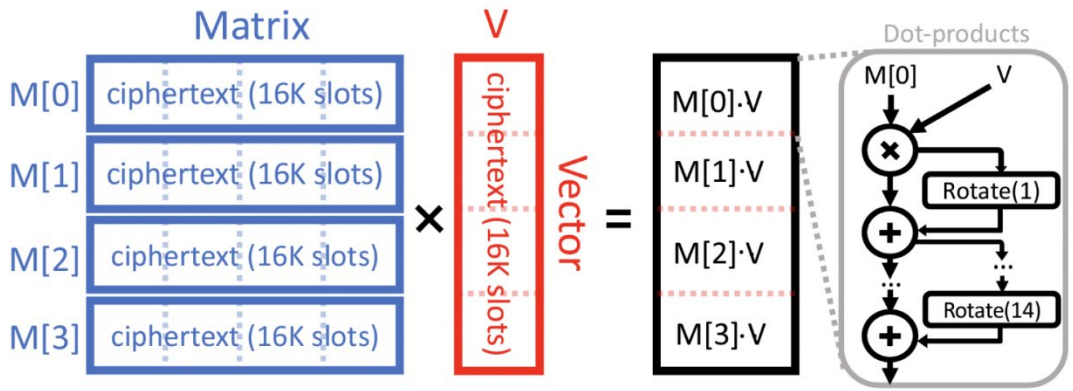
nverse NTT

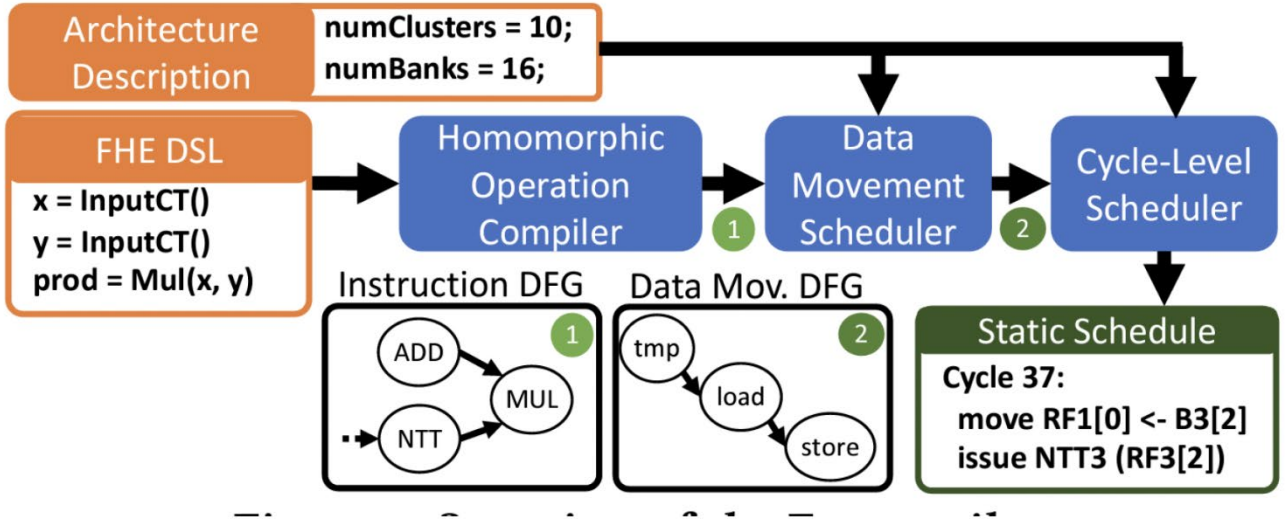
	T.W.	T
s	**15.4 μs**	**1.03×**
s	19.4 μs	0.91×
s	**26.6 μs**	**1.10×**
s	**50.1 μs**	**1.05×**
s	**91.1 μs**	**1.06×**
s	**20.5 μs**	**1.03×**
s	**28.6 μs**	**1.21×**
s	**49.9 μs**	**1.19×**
us	**93.9 μs**	**1.29×**
us	**173.6 μs**	**1.29×**
s	**24.1 μs**	**1.01×**
s	**53.2 μs**	**1.23×**

2^{14}	32	57.3 μs	**50.1 μs**	**1.14×**	64.5 μs	**51.8 μs**	**1.24×**	121.8 μs	**84.6 μs**	**1.44×**	143.3 μs	**97.2 μs**	**1.47×**
	64	112.6 μs	**96 μs**	**1.17×**	147.4 μs	**99.3 μs**	**1.48×**	218.5 μs	**160.5 μs**	**1.36×**	266.8 μs	**180 μs**	**1.48×**
	128	210.7 μs	**176.1 μs**	**1.19×**	277.1 μs	**183 μs**	**1.51×**	420.8 μs	**303.19 μs**	**1.38×**	511.9 μs	**331.7 μs**	**1.54×**
2^{15}	4	25.6 μs	26.4 μs	0.96×	28.7 μs	**25.9 μs**	**1.10×**	35.8 μs	41.2 μs	0.86×	47.3 μs	50.3 μs	0.94×
	16	64.1 μs	**52.2 μs**	**1.22×**	74.7 μs	**53.2 μs**	**1.40×**	147.1 μs	**100 μs**	**1.47×**	170.1 μs	**95.6 μs**	**1.78×**
	32	136.3 μs	**100.3 μs**	**1.35×**	173 μs	**102.4 μs**	**1.69×**	266.2 μs	**191.8 μs**	**1.38×**	325.5 μs	**193.2 μs**	**1.68×**
	64	254.9 μs	**192.1 μs**	**1.32×**	322.2 μs	**193.6 μs**	**1.66×**	514.2 μs	**372.2 μs**	**1.38×**	633.7 μs	**377.7 μs**	**1.67×**
	128	491.1 μs	**362.4 μs**	**1.35×**	623.1 μs	**364.3 μs**	**1.71×**	998.9 μs	**709.3 μs**	**1.41×**	1202.9 μs	**725.2 μs**	**1.66×**

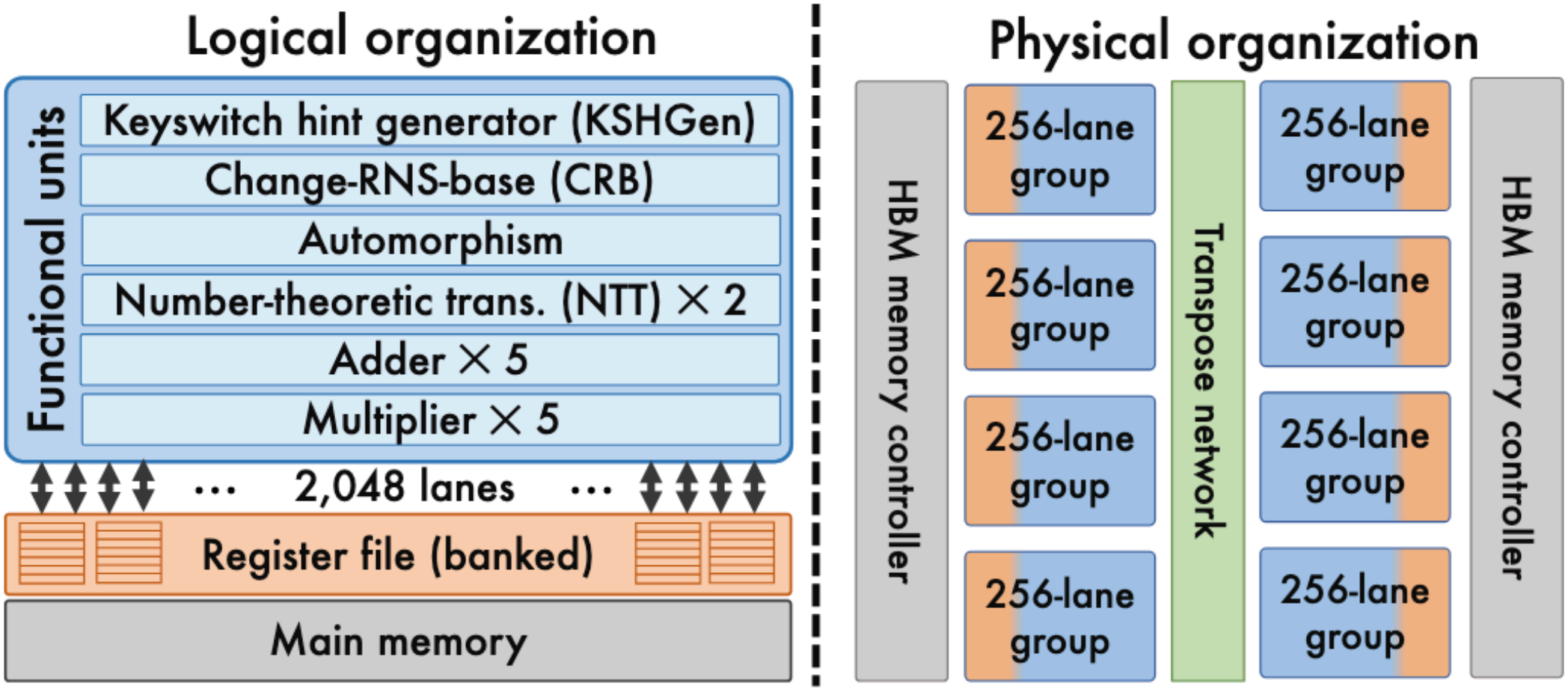
Specialist Hardware (F1)



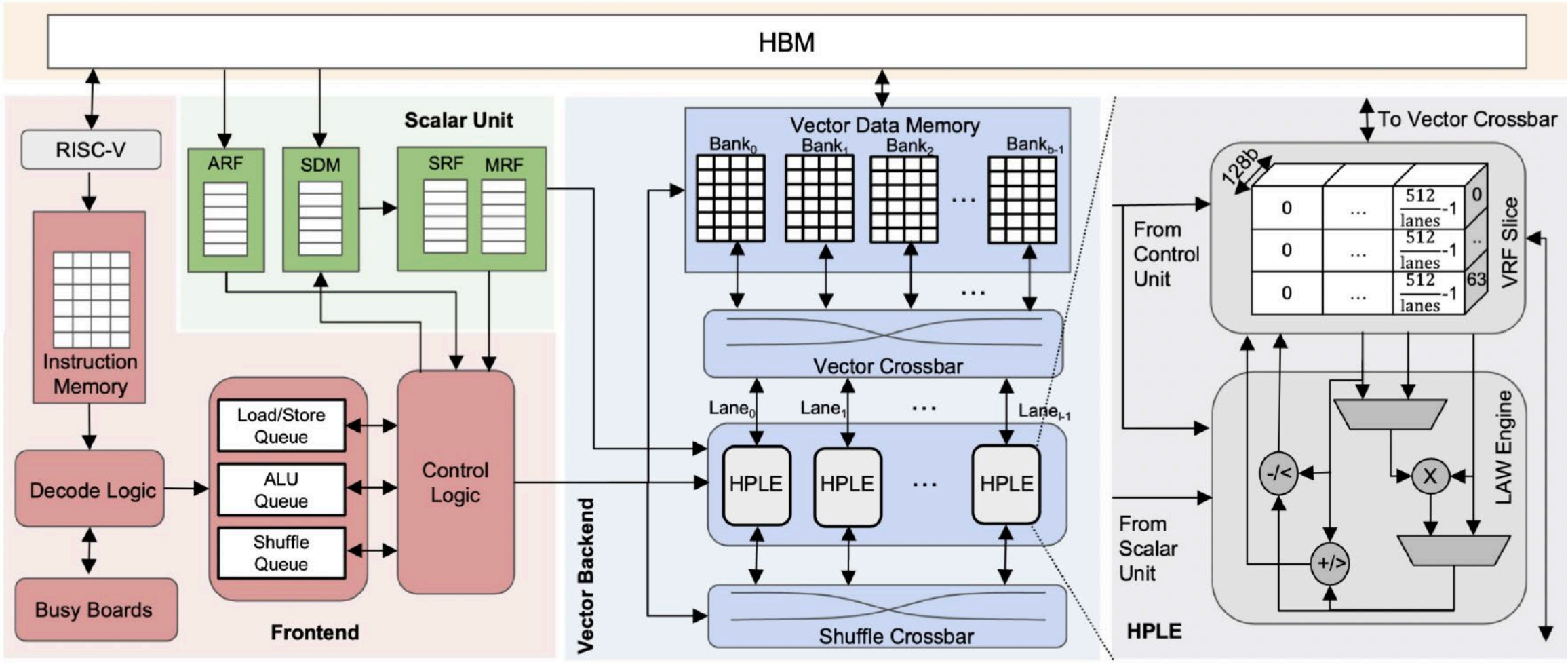




Specialist Hardware (Craterlake)



Specialist Hardware (RPU)



FHE:
10. Data Fragmentation

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

Fragmentation

10.2 Aggregation

With aggregation HE, we can split the data into a number of fragments, and then distribute these. Castelluccia et al [113] uses a parent and child architecture for an ad-hoc network. For this each node adds their routing hop data with homomorphic encryption. To encrypt a message(m) within a range of 0 to $M-1$, we generate a random keystream value of k (between 0 and $M-1$). To encrypt we compute:

$$c = Enc(m, k, M) = m + k(\text{mod } M) \tag{10.1}$$

and to decrypt:

$$Dec(c, k, M) = c - k(\text{mod } M) \tag{10.2}$$

For the addition of ciphertexts, we have:

$$c_1 = Enc(m_1, k_1, M)c_2 = Enc(m_2, k_2, M) \tag{10.3}$$

and simply add the ciphertext values:

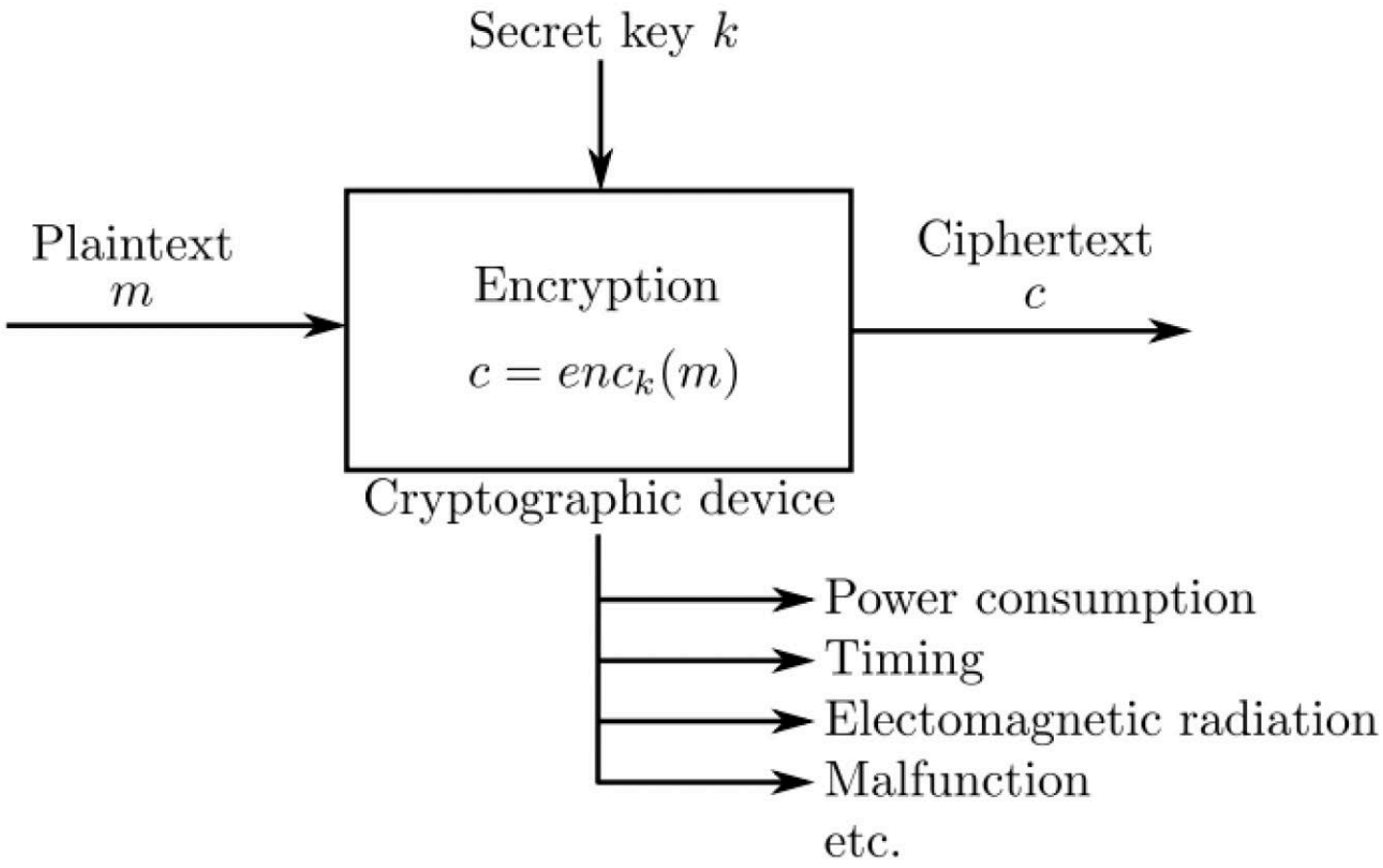
$$c_1 + c_2 = Enc(m_1 + m_2, k_1 + k_2, M) \tag{10.4}$$

FHE:
11. Side Channel

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

TBC


Secret key k
Plaintext
m
Encryption
$c = enc_k(m)$
Ciphertext
c
Cryptographic device
Power consumption
Timing
Electomagnetic radiation
Malfunction
etc.

Side-channel

11.3.1 Single-Trace Attack on SEAL's BFV Encryption Scheme

The study in [120] introduced the first single-trace side-channel attack on homomorphic encryption (HE), specifically targeting the SEAL implementation of the Brakerski/Fan-Vercauteren (BFV) scheme prior to v3.6. The attack exploits power-based side-channel leakage during the Gaussian sampling phase of SEAL's encryption process, enabling plaintext recovery with a single power measurement. The proposed attack follows a four-step methodology to recover plaintext messages from SEAL's BFV encryption scheme.

11.3.2 Single-Trace ML Attack on CKKS SEAL's Key Generation

The authors in [125] uncover new side-channel vulnerabilities in Microsoft SEAL by focusing on its number theoretic transform (NTT) function using power analysis. Specifically, it presents an attack targeting the NTT operation within the SEAL CKKS scheme's key generation process. The study demonstrates that the NTT, used during key generation, leaks ternary values $(-1, 0, +1)$ corresponding to secret key coefficients. The key innovation lies in developing a sophisticated two-stage neural network-based classifier capable of extracting side-channel information from a single measurement, demonstrating an unprecedented 98.6% accuracy in revealing secret key coefficients on the ARM Cortex-M4F processor. The research explores the impact of compiler optimizations, analyzing SEAL's NTT implementation across optimization levels from `-O0` (no optimization) to `-O3` (maximum optimization). While `-O3` eliminates previously identified vulnerabilities, the study reveals new side-channel leakages in the `guard` and `mul_root` operations under this setting. Random delay insertion, evaluated as a countermeasure, is shown

Side-channel

11.3.3 Side-Channel Vulnerabilities in LWE/LWR-Based Cryptography

The authors in [126] demonstrated successful attacks against multiple post-quantum cryptography implementations, breaking through various side-channel countermeasures, including masking and shuffling techniques. Their work revealed that these vulnerabilities are not implementation-specific but rather stem from core algorithmic properties of LWE/LWR-based cryptography. While this research does not explicitly target homomorphic encryption schemes, its findings have significant implications for the broader family of LWE/LWR-based cryptographic systems, including homomorphic encryption. As many modern homomorphic encryption schemes are built upon these same mathematical foundations, they could potentially be vulnerable to similar side-channel attacks targeting specifically, the incremental storage of decrypted messages in memory and ciphertext malleability properties inherent to LWE/LW based schemes.

11.3.4 Cache-Timing Attack on the SEAL Homomorphic Encryption Library

The authors in [127] exposed a cache-timing vulnerability in the SEAL homomorphic encryption library, specifically in its implementation of Barrett modular multiplication. The researchers identified a timing side-channel in the non-constant-time implementation of extra-reductions using the ternary operator, which leaks information about the secret key. Leveraging a novel remote cache-timing methodology, the attack aims to recover the secret key involved in modular multiplications. By analyzing ciphertexts that cause an extra reduction, the researchers solve Diophantine equations to progressively narrow down the range of possible secret keys. The approach combines insights from Bézout's theorem with optimized enumeration techniques to refine key candidates efficiently. The attack requires as few as eight ciphertexts that trigger extra reductions to uniquely determine the secret key. To

Side-channel

11.3.5 Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption

The authors in [128] introduced a single-trace side-channel attack targeting lattice-based cryptography, demonstrating its vulnerability to key recovery using observations from a single decryption. Unlike previous side-channel attacks, this approach is uniquely powerful because it can penetrate even masked implementations by recovering individual shares and subsequently reconstructing the complete decryption key. It focuses on the Number Theoretic Transform (NTT), a critical component in almost all efficient lattice-based cryptography implementations, making the attack applicable across a broad range of encryption schemes, including homomorphic encryption schemes. Unlike previous differential power analysis (DPA) attacks, which overlooked NTT, the authors leverages its less-protected nature. The attack

11.3.6 A Practical Full Key Recovery Attack on TFHE and FHEW by Inducing Decryption Errors

The study in [129] introduced the latest side-channel attack targeting Fully Homomorphic Encryption (FHE) schemes on the server side. Unlike prior attacks that focused on the client side, this study demonstrates that a malicious server can inject carefully calculated perturbations into ciphertexts stored on the cloud to induce decryption errors on the client side. The client, unaware of the malicious intent, reports these errors to the server. By analyzing the pattern of errors and the timing information associated with homomorphic operations, the attacker can gradually extract the underlying error values for each ciphertext. Specifically, these errors are used to reconstruct a system of linear equations that, when solved, compromise the security of the underlying Learning with Errors (LWE) problem and recover the client's secret key.

Side-channel (Mitigation)

11.4.1 Constant-Time Implementations

Constant-time implementations are a cornerstone of secure cryptographic design, addressing timing side-channel vulnerabilities by ensuring that operations execute in a consistent manner, independent of the input data or secret parameters [130]. Cryptographic algorithms and their implementations are crafted to eliminate timing variations that could inadvertently leak sensitive information, such as secret keys or computational states. This involves uniform execution patterns, where each branch of the code consumes the same computational resources and takes an equal amount of time to complete, regardless of the processed data. By adhering to constant-time principles,

11.4.2 Masking and Blinding Techniques

Masking and blinding are techniques that randomize intermediate values to mitigate side-channel attacks. When applied to symmetric block ciphers, this is referred to as masking, where data is split into randomized shares. In contrast, when used in public key cryptosystem implementations, it is called blinding, involving the addition of random noise to obfuscate correlations [132]. These techniques introduce random noise or perturbations during cryptographic operations to obscure the computational process and prevent attackers from correlating power traces, timing variations, or other measurable characteristics with secret data. By randomizing internal computations

Side-channel (Mitigation)

11.4.3 Shuffling and Randomization Techniques

Shuffling and randomization [133] techniques can play a crucial role in mitigating side-channel attacks in the context of Homomorphic Encryption (HE). These methods aim to obfuscate execution patterns and data processing sequences, complicating an attacker's ability to correlate observable side-channel data—such as power consumption, electromagnetic emanations, or timing variations—with sensitive cryptographic operations [134].

- **Shuffling:** Shuffling involves executing operations or processing data in a randomized order [133]. For example, in HE systems that handle multiple ciphertexts or compute on batched data, shuffling the order of operations can disrupt predictable patterns that attackers rely on to analyze side-channel data. This technique is particularly useful in thwarting statistical side-channel attacks, where repeated patterns are exploited over multiple traces to infer secret information.

- **Random Dummy Operations:** Inserting random, non-functional operations (dummy computations) during cryptographic processing adds noise to power or timing traces, making it harder for attackers to distinguish real computations [134]. In HE, dummy operations must be designed to avoid disrupting the correctness of computations, as the deterministic nature of HE schemes leaves little room for errors introduced by excessive obfuscation. Practical implementations need to balance security gains with the additional computational burden.

- **Randomized Noise Addition:** Adding random noise to intermediate values during HE operations can obscure side-channel data, reducing the risk of pattern detection. However, in HE systems, this noise must be carefully managed to avoid interfering with the decryption process or exceeding the noise budget inherent to HE ciphertexts. Unlike masking, which binds randomness to cryptographic parameters, randomized noise in HE should be lightweight and consistent with the system's correctness constraints.

- **Obfuscated Memory Access Patterns:** Randomizing memory access patterns prevents attackers from exploiting cache-timing or memory-based side-channel vulnerabilities [135]. In HE implementations, this could involve accessing memory blocks in a randomized order or using constant-time memory access strategies to eliminate data-dependent variations. For example, during ciphertext storage or retrieval, randomized access can ensure that memory usage does not reveal sensitive information, albeit at the cost of increased memory latency.